

NAVAL POSTGRADUATE SCHOOL

Monterey, California

2

AD-A245 738



DTIC
ELECTE
FEB 11 1992
S. D.

THESIS

THE SHORTEST PATH PROBLEM IN THE PLANE WITH
OBSTACLES: BOUNDS ON PATH LENGTHS AND SHORTEST
PATHS WITHIN HOMOTOPY CLASSES

by

Andre' M. Cuerington

June, 1991

Thesis Advisor:

J. R. Thornton

Approved for public release; distribution is unlimited

92-03288



REPORT DOCUMENTATION PAGE			
1a REPORT SECURITY CLASSIFICATION Unclassified		1b RESTRICTIVE MARKINGS	
2a SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.	
2b DECLASSIFICATION/DOWNGRADING SCHEDULE			
4 PERFORMING ORGANIZATION REPORT NUMBER(S)		5 MONITORING ORGANIZATION REPORT NUMBER(S)	
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (If applicable) 53	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	
8a NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS	
		Program Element No	Project No
		Task No	Work Unit Accession Number
11. TITLE (Include Security Classification) THE SHORTEST PATH PROBLEM IN THE PLANE WITH OBSTACLES: BOUNDS ON PATH LENGTHS AND SHORTEST PATHS WITHIN HOMOTOPY CLASSES			
12. PERSONAL AUTHOR(S) Andre M. Cuerington			
13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED From To	14. DATE OF REPORT (year, month, day) June 1991	15. PAGE COUNT 118
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
17 COSATI CODES		18. SUBJECT TERMS (continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUBGROUP	
		Path Planning, Finding the Shortest Path in the Plane With Obstacles	
19 ABSTRACT (continue on reverse if necessary and identify by block number) The problem of finding the shortest path between two points in a plane containing obstacles is considered. The set of such paths is uncountably infinite, making an exhaustive search impossible. This difficulty is overcome by reducing the size of the search space. The search is first restricted to a countably infinite set by focusing attention on the set of homotopy classes. By applying simple optimality principles, we obtain a finite list of such classes whose union contains the shortest path. This process of simplification is discussed in the thesis of CAPT Kevin D. Jenkins, U.S. Marine Corps. In this thesis we first discuss a computational investigation of two methods by which homotopy classes can be named. Next, a computational heuristic is presented that finds the lower bound for a path in a class. Finally, the true shortest path is found by searching these classes in order of increasing lower bound. One application of this study is in the area of robotic path planning.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS REPORT <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL J. R. Thornton		22b TELEPHONE (Include Area code) (408) 646-2741	22c OFFICE SYMBOL Ma/Th

Approved for public release; distribution is unlimited.

The Shortest Path Problem in the Plane With Obstacles:
Bounds on Path Lengths and Shortest Paths
Within Homotopy Classes

by

André M. Cuerington
Captain, United States Army
B.S., United States Military Academy, 1984

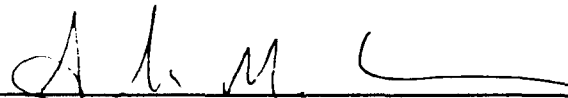
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

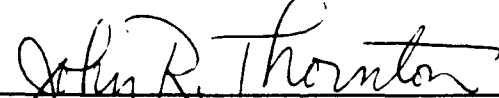
NAVAL POSTGRADUATE SCHOOL
June, 1991

Author:



André M. Cuerington

Approved By:



John R. Thornton, Thesis Advisor



Kim A. S. Query, Second Reader



Harold M. Fredricksen, Chairman,
Department of Mathematics

ABSTRACT

The problem of finding the shortest path between two points in the plane containing obstacles is considered. The set of such paths is uncountably infinite, making an exhaustive search impossible. This difficulty is overcome by reducing the size of the search space. The search is first restricted to a countably infinite set by focusing attention on the set of homotopy classes. By applying simple optimality principles, we obtain a finite list of such classes whose union contains the shortest path. This process of simplification is discussed in the thesis of CAPT Kevin D. Jenkins, U.S. Marine Corps. In this thesis we first discuss a computational investigation of two methods by which homotopy classes can be named. Next, a computational heuristic is presented that finds the lower bound for a path in a class. Finally, the true shortest path is found by searching these classes in order of increasing lower bound. One application of this study is in the area of robotic path planning.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

I.	INTRODUCTION	1
	A. THE PROBLEM	1
	B. OVERVIEW OF THE SOLUTION	1
	C. THE APPROACH	2
	1. The Topology	2
	2. Establishing the Naming Convention	3
	3. Generation of Names for Candidate Equivalence Classes	4
	4. Heuristic Ordering of Candidate Homotopy Classes	4
	5. Class by Class Solution of the Shortest Path Problem	5
	D. SUMMARY	5
	E. THE CONTRIBUTION	6
II.	NAMING EQUIVALENCE CLASSES	7
	A. INTRODUCTION	7
	B. CONSTRUCTION OF REFERENCE FRAME	8
	C. RAW CHARACTER STRINGS	9
	D. ALGORITHM 1	10
III.	A COMPUTATIONAL INVESTIGATION	13
	A. INTRODUCTION	13
	B. ALGORITHM 2	13
	1. Fundamental Group	13
	2. Functions in Algorithm	17
	a. Side Array	17

	b.	Switch Function	17
	c.	Index Function	17
	d.	The Algorithm	18
	e.	Fundamental Group Cancellation Function	19
C.		THE COMPUTATIONAL INVESTIGATION	20
	1.	The Approach	20
	2.	The Test	21
IV.		DETERMINING A BOUND FOR A HOMOTOPY CLASS	24
	A.	INTRODUCTION	24
	B.	FINDING THE BOUND	27
	1.	Establishing a Cone of Directions	27
	2.	Class Names Defining Cones	27
	3.	Intersecting Multiple Cones	28
	4.	An Empty Intersection	31
	C.	THE ALGORITHM	33
V.		FINDING THE SHORTEST PATH	35
	A.	INTRODUCTION	35
	B.	FINDING LINE SEGMENTS	36
	C.	DETERMINING THE CORRECT TANGENT LINES	41
	1.	Point to Obstacle	41
	2.	Obstacle to Obstacle	42
	D.	TWO POTENTIAL PROBLEMS TO CHECK	49
VI.		CONCLUSIONS AND RECOMMENDATIONS	56
	A.	CONCLUSIONS	56
	B.	RECOMMENDATIONS FOR FURTHER STUDY	58

APPENDIX A. FORTRAN PROGRAM	60
APPENDIX B. A SHORT EXAMPLE	102
LIST OF REFERENCES	108
INITIAL DISTRIBUTION LIST	109

ACKNOWLEDGMENTS

I fully acknowledge the assistance I received from Dr. John Thornton, Naval Post School Mathematics Department, and CAPT Kevin D. Jenkins, U. S. Marine Corps. The tremendous support of these two professionals was critical during my effort to complete this thesis.

Dr. Thornton's staunch guidance inspired the motivation to start this work and the determination to finish it. In addition to other technical assistance, Dr. Thornton established topological proofs which support the theories behind the algorithms presented in Chapters II and III.

Much of the research for this project was conducted in conjunction with the thesis research of CAPT Jenkins. We spent many months together writing and debugging the FORTRAN program which is presented in Appendix A. Following the joint research and development of the shortest path problem, Chapter I (with the exception of minor changes) was written by CAPT Jenkins for inclusion in this thesis.

I extend much thanks to these two individuals.

I. INTRODUCTION

The industrial community relies today, in increasing measure, upon robots to handle a multitude of tasks. A need remains to develop robots, and fixed robot arms, with the ability to roam freely among obstacles. In this regard, the calculation of shortest paths is of obvious importance.

A. THE PROBLEM

As robot arms move, they must vary their configurations to position their end-effector or "hand". Once obstacles are introduced, freedom of movement of the manipulator may become drastically reduced. The problem considered here is one of finding the shortest path from a starting point to a destination point in a planar region containing obstacles. This paper addresses several parts of the solution to this problem.

B. OVERVIEW OF THE SOLUTION

Between any two points in the plane, a path joining them may be chosen from an uncountably infinite set of alternatives. It is desirable to choose the shortest path from this set which, due to the presence of obstacles, may not be a straight line. An exhaustive search of the collection of possible paths is impossible, so another method must be developed.

The set of possible paths is partitioned into a collection of equivalence classes--mutually exclusive subsets which collectively exhaust the partitioned set. This partition is induced when an equivalence relation is defined on the set of possible paths. This set of equivalence classes is countably infinite.

Naming conventions are established which associate with each class a character string which allows the classes to be represented in a computer.

Next, a finite list of candidate classes is produced to search.

A heuristic is then applied to this finite list of classes to place them in an order which saves computational effort.

The final step begins with this ordered list of candidate classes and solves the shortest path problem class by class in listed order. Savings of computational effort are realized when the class-by-class solution process can terminate without exhausting the ordered list of candidate classes.

The key ideas in the above overview are now considered in somewhat greater detail.

C. THE APPROACH

1. The Topology

Let P be the uncountably infinite set of all continuous, obstacle-avoiding paths from a starting point to a destination point, denoted a and z respectively. If p_i and

p_i and p_j are two paths in P , which have the same starting point and destination point, we say that p_i is homotopic to p_j if p_i can be mapped to p_j under a continuous function (with both endpoints fixed in place) without encroaching on any obstacles [Ref. 1:p. 223]. Clearly, the homotopy relation is reflexive, symmetric and transitive and therefore defines an equivalence relation [Ref. 1:p. 223]. This relation induces a partition of P into a countably infinite collection of equivalence classes, known as homotopy classes [Ref. 1:p. 223].

2. Establishing the Naming Convention

In order to name homotopy classes, a reference frame is established to represent the topological relationships in the plane with obstacles. For a given path, p , a string of characters, $P(p)$, is recorded which encodes information concerning the relationship of the path to the obstacles.

Two algorithms are then presented which accept $R(p)$ as input. These algorithms have the following important property which implicitly defines names for the homotopy classes: If p_i and p_j are coterminal paths and $R(p_i)$ and $R(p_j)$ are input to either of the two algorithms, then the outputs are identical if and only if p_i is homotopic to p_j [Ref. 2].

Chapter III presents a computational investigation which closely examines the methods employed by these two algorithms to produce the names for these homotopy classes.

3. Generation of Names for Candidate Equivalence Classes

An edge-labeled graph is constructed which models the topological relationships within the region, where the nodes represent subregions induced by the reference frame. Nodes are connected if their corresponding subregions are adjacent. The names of the desired homotopy classes may be obtained by traversing this graph. This traversal produces a list of candidate classes which contain all paths of minimal length.

The location of the origin affects the manner in which the plane is divided into wedges. In this regard, the graph which is created, given the location of the origin in one instance, may not be the same as the graph obtained if the origin is moved to a new point.

4. Heuristic Ordering of Candidate Homotopy Classes

For each class on the list, a lower bound on the length of its shortest representative path is constructed. The list of classes is then arranged into increasing order of these bounds. To obtain the bounds, a point is first chosen within each obstacle. A contraction deformation is then be applied to each obstacle to "shrink" the obstacle to that chosen point.

Now some class is fixed and its shortest path is examined. As all obstacles are simultaneously contracted to their representing points, this shortest path has a limiting position which is polygonal. The length of this polygonal path is the lower bound which is associated with that class.

This length can be calculated from the class name and the representing points without explicitly defining any contraction deformation. This thesis presents this method to determine a bound for the shortest path in a given class.

5. Class by Class Solution of the Shortest Path Problem

In the final step of the solution, the classes on the ordered list of candidates are considered. The first class is removed from the list--that class with the smallest associated bound. The true shortest path in the named class is found by reversing the contraction previously applied to the obstacles, thereby transforming the polygonal path whose length provided the lower bound into the true shortest path. If the length of this path is smaller than the bound associated with the class on the top of the remaining list, the search is stopped. Otherwise, the first class is removed from that remaining list and the procedure is repeated. This procedure, which is contained in the thesis, continues until the condition specified above is met, and the shortest path has been found.

D. Summary

The solution to the problem of searching for the shortest path between two points in the plane with obstacles begins with consideration of a set of paths which is uncountably infinite. Through the homotopy relation, this set is reduced to a countably infinite set. The new set is further reduced to a finite list by modeling the region with a graph and

applying an optimality principle. This final list contains only those homotopy classes containing paths which are not self-crossing. From this list a shortest path is found.

Computational effort is further reduced through the use of a heuristic which orders the list of candidate classes by increasing order of their lower bounds. The heuristic used also facilitates a methodical search of the classes while solving the shortest path problem. The use of this heuristic does not, however, imply that the solution is approximate. The solution to the problem will be exact using this method.

E. THE CONTRIBUTION

Chapter II of this thesis presents the methods used to establish a reference frame given the plane with obstacles and an algorithm which is used to generate the homotopy class name of a given path.

Chapter III presents a computational investigation which addresses the question of whether or not two algorithms which are used to name homotopy classes actually provide the same results.

Chapter IV introduces a method to find a lower bound for the shortest path in a class.

Chapter V shows how to find the true shortest path in a class.

II. NAMING EQUIVALENCE CLASSES

A. INTRODUCTION

In this chapter, a labelling scheme is established to represent the equivalence classes. This notation is used throughout the paper to organize the computational search for the shortest path.

An algorithm is presented that for a character representation of a path, p , names the path's respective equivalence class. It can be shown that, after being processed by the algorithm, different character representations of paths p_i and p_j yield the same output exactly when and only when the two paths are in the same equivalence class [Ref. 2]. Thus, the name of a class will be the string obtained by applying the algorithm to any representative in the class.

The computation of class names depends on a reference frame which in turn depends on a collection of obstacles. Although more than one reference frame can be drawn for a particular collection of obstacles, the choice of a particular frame fixes the representation of all homotopy classes.

Once a reference frame is developed, the name of the homotopy class for a path can be determined by a two-step procedure. First, a character string $R(p)$ is calculated which encodes certain information about the path taken through the

obstacles. Second, this character string is accepted as input to an algorithm which then produces a name for the equivalence class to which the particular path belongs. This algorithm produces the class names in terms of the same alphabet used to create the initial character string. The results from the algorithm described, Algorithm 1, are used throughout this analysis as the class name associated with a given path.

B. CONSTRUCTION OF REFERENCE FRAME

Let b_k be an arbitrary point in obstacle B_k , $k = 1, 2, \dots, n$, where n is the number of obstacles in the region. A point c is chosen and a line drawn through each b_k , infinite in extent in each direction and having the following properties: there is an open disk, δ , centered at c such that $\delta \cap B_k = \emptyset$ for all $k=1, \dots, n$, and the n lines connecting c with the points b_k are distinct. Such a point c can always be found as the above two conditions are satisfied by any point in the planar region that is neither on an obstacle nor on the $n(n-1)/2$ lines determined by pairwise choices of distinct b_k .

To draw a reference frame, the n lines are first constructed joining c with each b_k . The line from c to b_k is labeled as L_k . Each line is then partitioned into two directed, semi-infinite rays and one finite length line segment. The ray directed from c in the direction away from b_k is called α_k . The ray emanating from b_k and away from c is denoted β_k . The remaining line segment, $[c, b_k]$, is also

denoted α_k . The reference frame is the collection of line segments and rays so constructed, as illustrated in Figure 2.1.

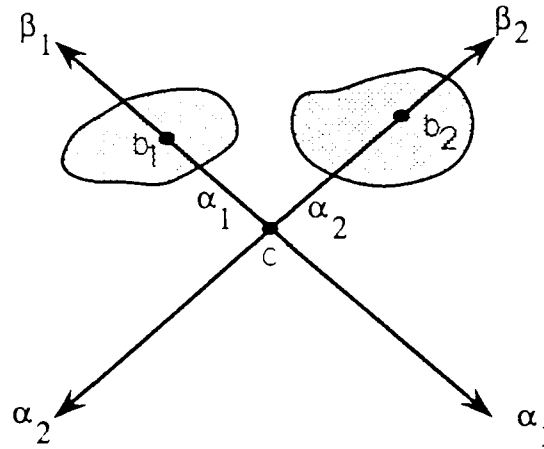


Figure 2.1 A Reference Frame With $n=2$ Obstacles

C. RAW CHARACTER STRINGS

A reference frame is fixed in a region T . Let a and z be points in T and let p be a directed path in T from a to z . Then the raw string of p , denoted $R(p)$, is defined to be the ordered sequence of characters obtained by following p from a to z and recording, in order, the names of the rays that are crossed.

Two special cases must be addressed so as to make the above definition complete. First, in the case that p crosses no reference rays, we let $R(p) = e$, where e denotes the empty string. Second, if p crosses through c (simultaneously

crossing all α_j) the names of all α_j will be recorded in order of increasing subscript.

Thus, raw strings are of the form

$$R(p) = x_1 x_2 \dots x_m$$

where each x_j belongs to the alphabet

$$A = \{e, \alpha_1, \alpha_2, \dots, \alpha_n, \beta_1, \dots, \beta_n\}.$$

The character x_j ($j = 1, 2, \dots, m$) is the name of the j^{th} reference ray crossed by p . Figure 2.2 shows a pair of paths connecting the two points a and z in the region with two obstacles.

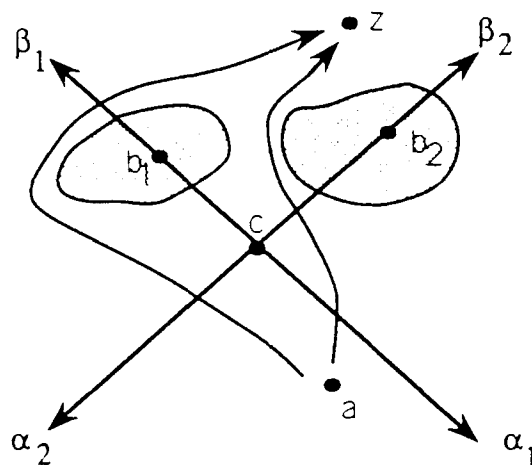


Figure 2.2 Paths $\alpha_1 \beta_1$ and $\alpha_2 \beta_2$

D. ALGORITHM 1

Algorithm 1 accepts as input a raw string $R(p)$ for any path p in T and any reference frame. The output is a string denoted $C(R(p))$ of characters also chosen from the alphabet A .

This output string $C(R(p))$ is the name of the homotopy class to which p belongs.

The algorithm is presented in terms of two functions. The first is the sorting function σ . Let $S = x_1 \dots x_m$ be any string over A . If S contains a two character substring $x_i x_{i+1} = \alpha_j \alpha_i$ with $i < j$, then $\sigma(S)$ is the string which results by reversing the order of the leftmost such substring. Figure 2.3 depicts two such strings and makes clear that such paths are homotopic. If S contains no such two character substring, then $\sigma(S) = S$.

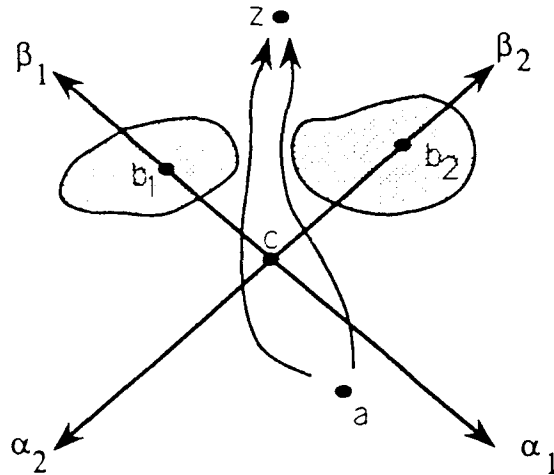


Figure 2.3 Homotopic Alpha Strings $\alpha_2 \alpha_1$ and $\alpha_1 \alpha_2$

Repeated application of σ sorts all substrings that consist entirely of α_j 's into non-decreasing order of subscript. $\Sigma(S)$ is defined to be the string which results when this sorting is complete.

The second function is the cancellation function χ . If string S contains a two character substring $x_i x_{i+1}$ with $x_i = x_{i+1}$

then $\chi(S)$ is the string which results by removing the leftmost such two character substring. Otherwise $\chi(S) = S$. Such cancellation is intuitive, for a reference ray that is crossed twice consecutively is equivalent to one that is not crossed at all. Thus repeated application of χ cancels all pairs of adjacent like occurrences of identical characters. Let $X(S)$ be defined as the string which results when all such possible cancellation is complete.

With these definitions complete, Algorithm 1 is given in Figure 2.4.

```

begin
k ← 0
Sk ← R(p)
WHILE Sk not equal to X(Σ(Sk))
    k ← k+1
    Sk ← X(Σ(Sk-1))
END WHILE
C(R(p)) ← Sk
END

```

Figure 2.4 Algorithm 1

The output from Algorithm 1, $C(R(p))$, is called the canonical representation of p and is the unique name for the homotopy class of p [Ref. 2]. That name is used throughout this paper.

III. A COMPUTATIONAL INVESTIGATION

A. INTRODUCTION

This chapter describes a computational investigation which was conducted as an aid to verifying that the homotopy class names produced by Algorithm 1 are equivalent to those produced by a known method of naming homotopy classes [Ref. 2].

A second algorithm, Algorithm 2, is introduced here to forge a link between the class names of Chapter II and the well-known fundamental group. Two processes are presented to exploit this link: If Algorithm 1 truly names the classes, then both procedures should produce the same output for every path tested. One million test cases were examined and no counter-examples were found. While this computational evidence is not a proof, it did provide initial support before the proof of the claim that Algorithm 1 names homotopy classes was found.[Ref. 2]

B. ALGORITHM 2

1. Fundamental Group

The class name obtained from Algorithm 1 was written in terms of the alphabet A defined in Chapter II. We present an algorithm here where the name obtained is expressed in terms of the fundamental group of a topological space T .

The basic idea for producing a fundamental group for this problem is to regard the paths in T as elements of the group and path concatenation $(*)$ as the group operation. There are two problems with considering the paths as the group elements. First, it is not necessarily possible to concatenate any two paths in T . In order to concatenate two paths using the operation $(*)$, the first path must end at the point where the second begins. In this analysis the test paths all begin and end at the point a . This point is called a base point and concatenation is possible for every pair of paths.

A second issue is that an inverse is not well defined for paths. If the identity is defined as the empty string, i.e., stay at the base point, then when a path and its inverse are concatenated we have traced out a path which is not equal to the identity. To avoid this problem, the homotopy classes of paths are considered. Then all paths that have an empty character string representation will be in the identity class. Also, any class followed by its inverse will equal the identity. In Figure 3.1, g_1 represents the class $\alpha_1\beta_1$, and $(g_1)^{-1}$ represents the class $\beta_1\alpha_1$. So, $g_1 * (g_1)^{-1}$ represents the class $\alpha_1\beta_1\beta_1\alpha_1$ which reduces to the identity class by the rules of Algorithm 1.

The group elements, therefore, are homotopy classes of loops around obstacles based at a common point and the group

operation is defined in terms of concatenation (*) of these classes.

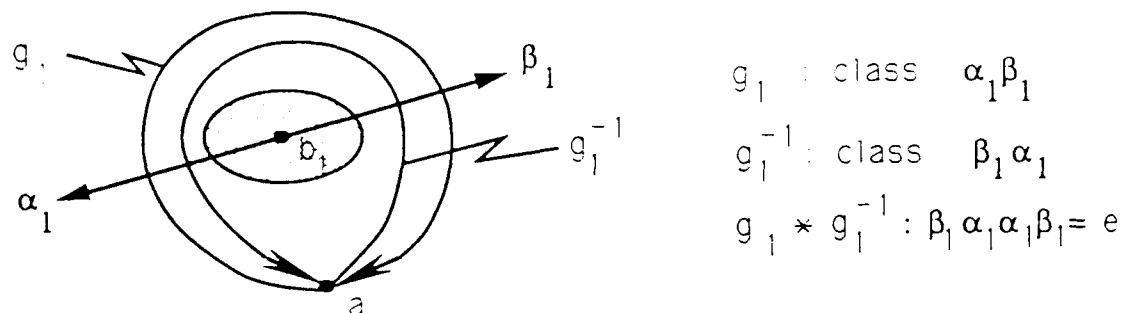


Figure 3.1 A Homotopy Class and its Inverse

Given a base point a , we let p be any loop beginning and ending at a . We let $[p_i]$ denote the homotopy class of p_i . So the set of group elements is $\{[p_i] \text{ such that } p_i \text{ is a member of } T, \text{ where } p_i \text{ is a loop based at } a\}$. This set will be called G with the elements denoted g_j . Figure 3.2 illustrates several paths and their homotopy classes given $n=2$ obstacles.

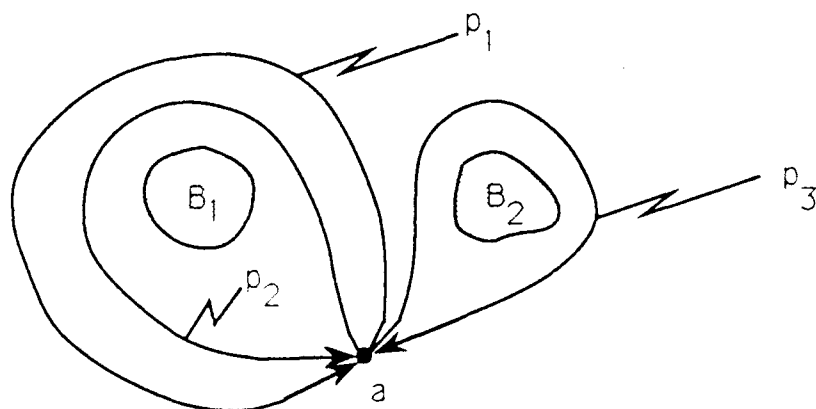


Figure 3.2 $[p_1] = [p_2] * [p_3]$

Then the group operation (*) can be defined by

$$[p] * [q] = [pq].$$

It is important to note that the fundamental group is finitely generated. As Figure 3.3 illustrates, the generators are not unique. However, by fixing a set of generators, the class names become fixed.

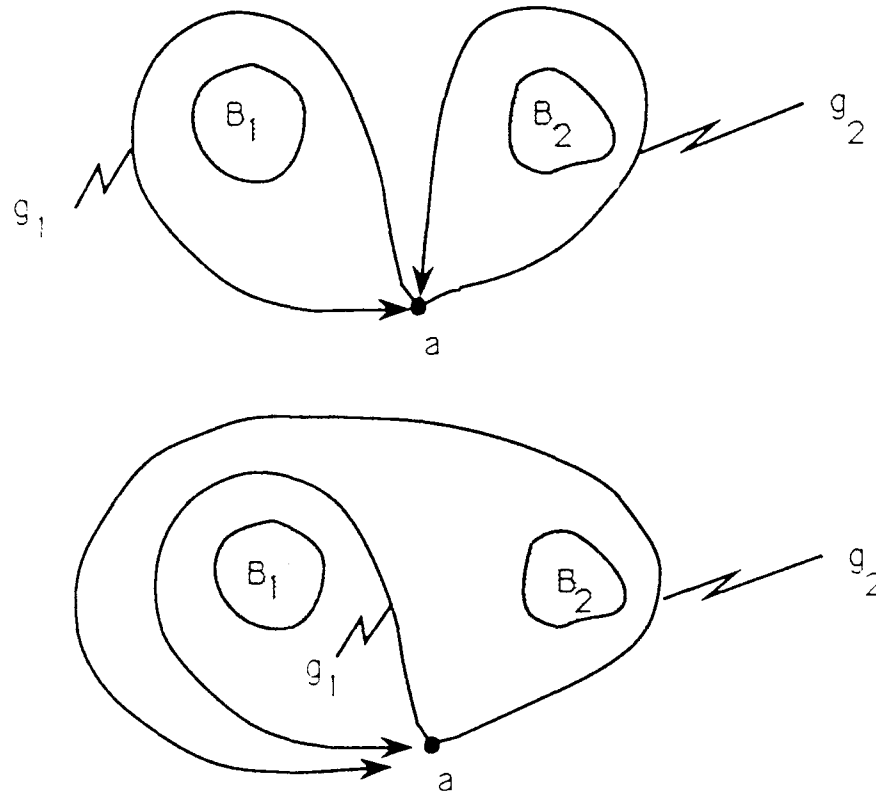


Figure 3.3 Alternate Generators For Fundamental Group of the Space With Two Obstacles

With the above information describing the fundamental group representation and a few necessary definitions, Algorithm 2 is given below.

2. Functions in Algorithm

a. Side Array

Consider the reference lines L_k constructed in Chapter II to be oriented rays with direction from c to b_k . It then becomes reasonable to discuss a 'right' and a 'left' side of those lines. A moving point is allowed to trace a path p from start point to destination point. $\text{Side}(k)$ is a function which defines the side of L_k on which the moving point lies. The output is either 'left' or 'right'. The output is never 'on' because this routine is used only after a complete crossing of L_k takes place. A crossing is considered to be complete when the moving point leaves L_k to one side after having met L_k from the opposite side.

b. Switch Function

The switch function is defined by:

$$\text{switch}(\text{side}(k)) = \begin{cases} \text{'right' if side}(k) = \text{'left'} \\ \text{'left' if side}(k) = \text{'right'} \end{cases}$$

As the moving point traces p , each time a reference line L_k is crossed, the switch function will be applied to indicate on which side of L_k the moving point lies.

c. Index Function

Let x_k be any character from the alphabet A representing the k^{th} element in $R(p)$. Let $\text{index}(x_k) = j$ if x_k equals α_j or β_j . Thus, the k^{th} crossing completed by the moving point is a crossing of L_j . So if the moving point is on the

left side of L_3 initially and this line is now crossed on either side of b_3 , then $\text{side}(3)$ would equal 'right'.

d. The Algorithm

Using the functions described above, Algorithm 2 is given in Figure 3.4. The algorithm works in two phases. The first phase initializes the array $\text{side}(k)$ for $k = 1, \dots, n$. The second phase reads the raw string from left to right and adds an element to the fundamental group representation for each β crossing. When the end of the input string is reached, the output is a shorter string with one character corresponding to every β element in the original raw string. Each character represents a generator or its inverse.

```

begin
input  $R(p) = x_1 x_2 \dots x_m$ 
for  $j = 1, \dots, n$ 
  if  $a$  is to right of reference line  $L_j$  then
     $\text{side}(j) \leftarrow \text{right}$ 
  else
     $\text{side}(j) \leftarrow \text{left}$ 
  end if
end for
 $G \leftarrow [a]$ 
for  $k = 1, \dots, m$ 
   $i = \text{index}(x_k)$ 
   $\text{side}(i) \leftarrow \text{switch}(\text{side}(i))$ 
  if  $x_k = \beta_r$  for some  $r$ , then
    if  $\text{side}(\text{index}(x_k)) = \text{left}$ , then
       $G \leftarrow G * g_i$ 
    else
       $G \leftarrow G * (g_i)^{-1}$ 
    end if
  end if
end for
 $F(R(p)) \leftarrow G$ 
end

```

Figure 3.4 Algorithm 2

e. Fundamental Group Cancellation Function

The cancellation rules in the fundamental group differ somewhat from those for the raw string. Although sets of generators are not unique, we can obtain a unique representation of each class with respect to a particular set of generators. For any given set of generators, every homotopy class can be represented as a product of these generators and their inverses. Even so, this representation is only unique after cancellation is applied. The cancellation rule follows.

Let $G = \{g_1, \dots, g_n\}$ be a set of generators of the fundamental group of T (base point a) and let $Y = y_1 y_2 \dots y_m$ be a representation of some homotopy class in terms of the g_j and their inverses, i.e. for each $i = 1, \dots, m$, $y_i = g_j$ or $(g_k)^{-1}$ for some $j, k = 1, \dots, n$.

The cancellation function κ is defined as follows. If Y contains a two character substring $y_i y_{i+1}$ with $y_i = (y_{i+1})^{-1}$ then $\kappa(Y)$ is the string which results by removing both y_i and y_{i+1} , in their leftmost occurrence, from Y . Finitely many repeated applications of κ produce a string in which no further cancellation is possible. We define this string as $K(Y)$.

C. THE COMPUTATIONAL INVESTIGATION

1. The Approach

A graphical representation of the two procedures is provided in Figure 3.5. The investigation accepts a random path p as input and generates two names for the homotopy class representing p as output.

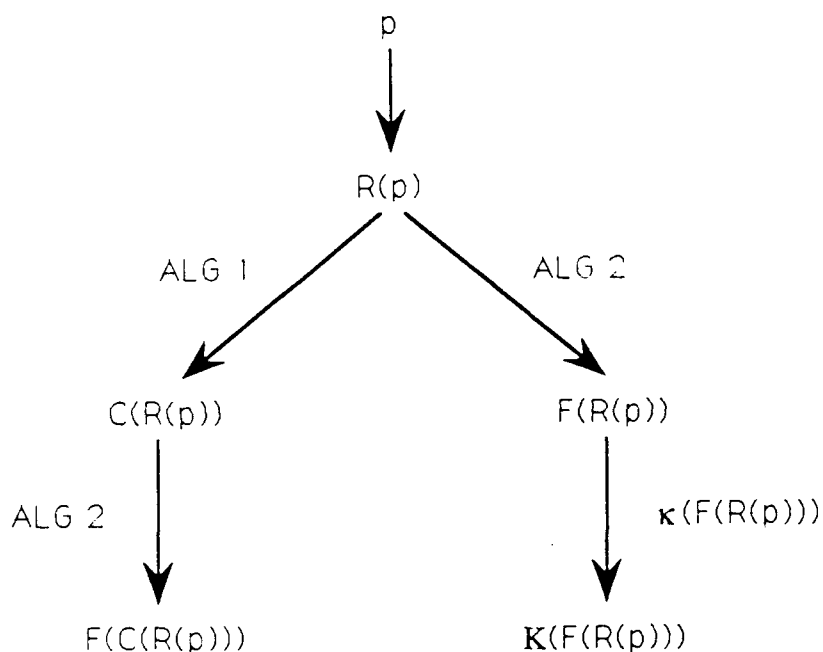


Figure 3.5 Flow Chart of Two Algorithms

The idea of the test is to determine for a given path its raw string character representation. For computational purposes, the paths considered are polygonal paths. The raw string is then used as input to both algorithms.

The raw string $R(p)$ is first input to Algorithm 2. The output $F(R(p))$ from Algorithm 2 is then input to the cancellation routine and the fully cancelled string $K(F(R(p)))$ is output. In parallel, $R(p)$ is input to Algorithm 1 which produces the canonical string $C(R(p))$. Algorithm 2 is then applied to $C(R(p))$ to output $F(C(R(p)))$. The results of these tests are then compared. In Figure 3.5 a flow chart illustrates this process. The program continues to compare the two outputs until a predetermined number of paths are checked.

It can be shown that $C(R(p))$ is a unique class name if and only if $F(C(R(p)))$ equals $K(F(R(p)))$ for all p in T [Ref. 2]. Before the proof of this conjecture was obtained, the above procedures were programmed and successfully tested for one million different paths.

2. The Test

The general algorithm used to test the two procedures represented in Figure 3.5 is easily followed and is shown in Figure 3.6. However, the attached program, which was used to test the model, is substantially more involved. This more involved program was written in an attempt to generate the most efficient programming code possible.

```

n = number of reference frames to be considered
m = number of paths to be considered
nobs = number of obstacles on each board

begin
for board = 1,..., n
  create reference frame or board
  for path = 1,..., n
    1. Create a polygonal path p with nseg
       segments
    2. Form R(p)
    3. Form F(R(p))
    4. Form K(F(R(p)))
    5. Form C(R(p))
    6. Form F(C(R(p)))
    if K(F(R(p))) does not equal F(C(R(p))) then
      print (raw string, board, and path info
            for the counter example)
    end if
  end for
end for
print (final seed)
end

```

Figure 3.6 Test Algorithm

Two time saving techniques are employed in order to test the one million cases in 25 minutes. The first method involves the subroutines which perform cancellation. It is simple to code a program that scans the character representation repeatedly to find all possible cancellation in a string such as $\beta_1\beta_2\alpha_1\alpha_1\beta_2$, which reduces to β_1 . However, in the enclosed code (Appendix A), pointers are inserted in the strings to mark the position where the first cancellation occurs. Then the newly adjacent characters are checked for possible cancellation. This method reduces the complexity of the test and is employed in both Algorithm 1 and Algorithm 2.

The second time-saving device was developed around the sorting of character strings. Instead of using a bubble sort method (order n^2 complexity), a merge sort algorithm (order $n(\log(n))$ complexity) is used. Table 1 shows the savings is realized immediately as opposed to there existing some cut off where $n(\ln(n))$ becomes less expensive.

TABLE 1		
COMPLEXITY COMPARISON		
n	n^2	$n(\ln(n))$
2	4	1
5	25	8
10	100	23
100	10,000	460
1,000	1,000,000	6,907
1,000,000	10E16	13,815,510

As can be seen in the attached code, this choice to save computer time requires a significant increase in program complexity and therefore in programmer time. The trade-off of computer time versus programmer time needs to be considered in any similar analysis.

The code used to implement the test is presented in Appendix A.

IV. DETERMINING A BOUND FOR A HOMOTOPY CLASS

A. INTRODUCTION

For each homotopy class on a list, a lower bound for the length of its shortest representative path is found. The list is obtained through a graph traversal process [Ref. 3]. The list of classes is then arranged in increasing order based upon these bounds.

To obtain the bounds, we first fix a class and consider its shortest path, p . For example, consider the path in Figure 4.1 to be a string attached at z . The shortest path is the one obtained by pulling the string tight so that it lies against the obstacles and passes through "a".

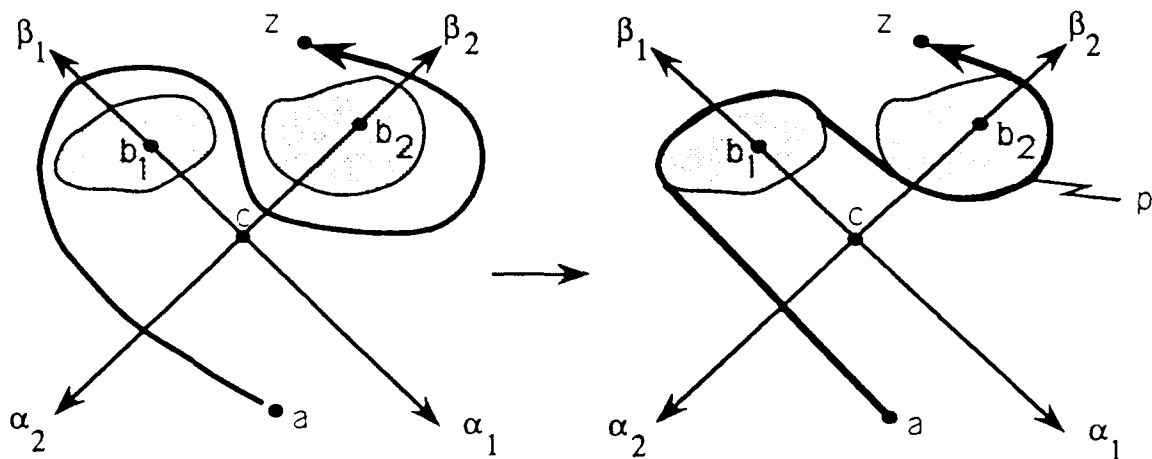


Figure 4.1 Class Representatives

The topological deformation of "contraction" may then be applied to each obstacle to "shrink" the obstacle to an point

b_k which is chosen arbitrarily within the obstacle B_k [Ref. 4]. As seen in Figure 4.2, when all obstacles can be simultaneously contracted to b_k . The shortest path p^* found in this way has a limiting position which is polygonal.

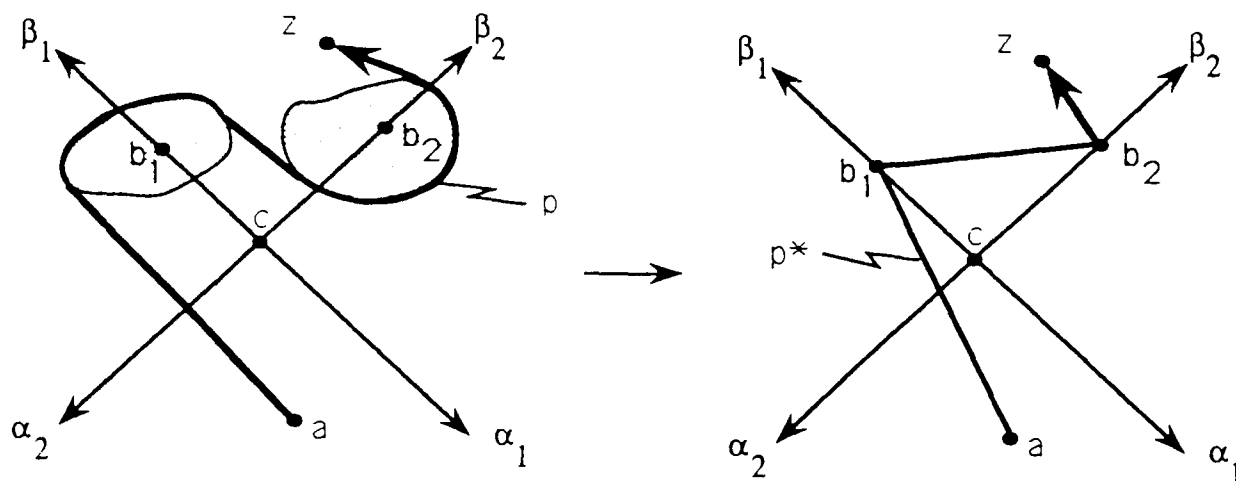


Figure 4.2 Class Representative to a Bound for the Shortest Representative

The length of this polygonal path provides a lower bound that is associated with that class. The fact that this is a lower bound follows from the following argument. If we let D be the set of points through which paths may travel with full-sized obstacles and E be the set of available points in the space after the obstacles are "shrunk", then D is a subset of E . Let $\Pi(X, E)$ be the set of all paths in equivalence class X which remains entirely in E . Since D is a subset of E , $\Pi(X, D)$ is a subset of $\Pi(X, E)$ for all X . We now define a function f that maps $\Pi(X, E)$ to a real number in \mathbf{R} , with the rule that

$f(p)$ = length of p . Since D is a subset of E , this equation follows:

$$\min_{p \in \Pi(X, E)} f(p) \leq \min_{p \in \Pi(X, D)} f(p) \quad (\text{Figure 4.3}) .$$

This lower bound, $\min_{p \in \Pi(X, E)} f(p)$, can be calculated from the class name and obstacle-representing points without explicitly defining any contraction deformation.

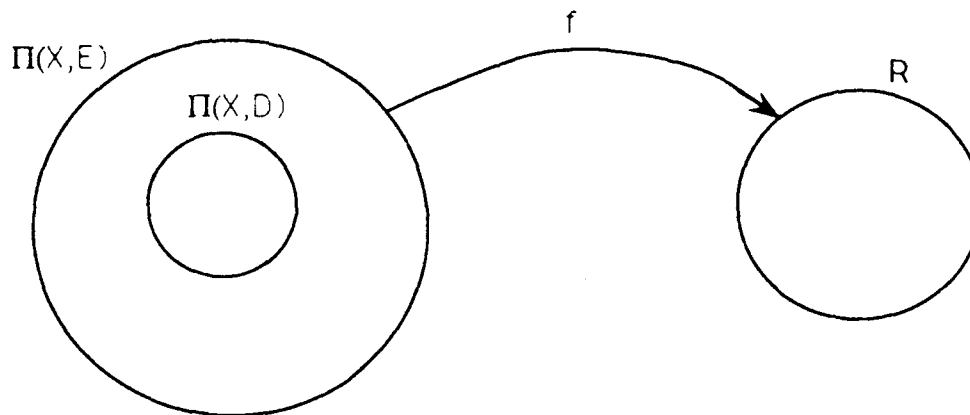


Figure 4.3 Mapping Related to Class Lower Bound

Knowing that this polygonal path, p^* , exists, we now give a method to find the minimum length path and compute its length. Effectively, we must connect the points b_k in the correct order and compute the length of each line segment.

The idea employed to find p^* is: We are given a starting point "a" and a character string which represents the mandatory order of ray crossings. Each character of the string implies that we must cross a certain ray. We then compute the maximum number of rays (i.e. read the maximum

number of characters in the string) that can be crossed with a straight line. If z cannot be reached with a straight line we determine where it is best to make a turn. The turning point then becomes the new starting point and the process repeats until z is reached. To find the straight line segments of p^* , the notion of a cone of directions in which the segments of a polygonal p^* must lie is introduced.

B. FINDING THE BOUND

1. Establishing a Cone of Directions

The cone of directions as depicted in Figure 4.4, is an open region--not including the boundaries--formed by two rays based at a common point and forming an angle of less than π radians. In this region we can draw any ray based at the point common to the boundary rays. This is the cone of directions.

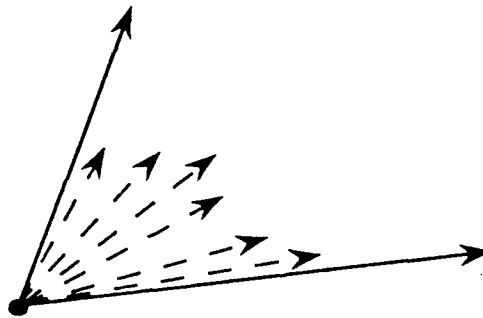


Figure 4.4 Cone of Directions

2. Class Name Defines the Cone

To create a cone we need a starting point "a" and a reference ray (Figure 4.5). We define an entire class name as α_k for a path to cross the ray α_k . First we draw a ray from

the starting point to the point representing the obstacle point b_k . In Figure 4.5, to cross α_k we must be to the right of the ray from "a" to b_k . The ray from "a" to b_k is one boundary for the cone. Next we draw a ray parallel to the ray that we must cross, α_k . Again in Figure 4.5, in order to cross α_k with a straight line we must leave "a" at an angle that will allow us to cross the α_k ray. Therefore, the ray parallel to α_k forms the other boundary. The two boundaries form the cone of directions as any ray strictly lying between these two limits will cross α_k .

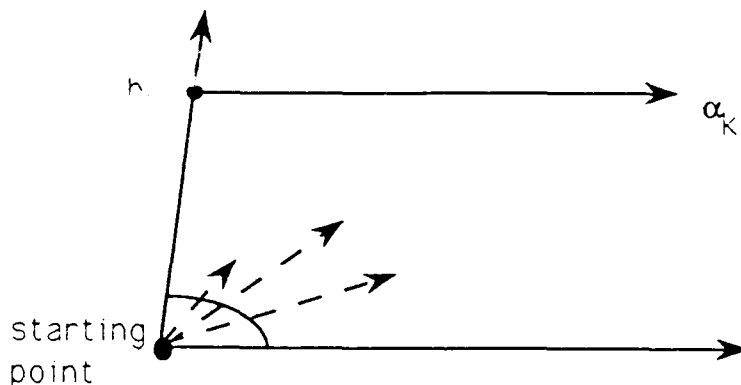


Figure 4.5 A Cone of Directions Toward α_k

3. Intersecting Multiple Cones

Given a cone of directions for a single ray, we can then consider longer class name strings. For this we generate an intersection of multiple cones of direction. For example, in Figure 4.6, "a" is the starting point and the class is $\alpha_1\beta_1\beta_2$ leading to the ending point z.

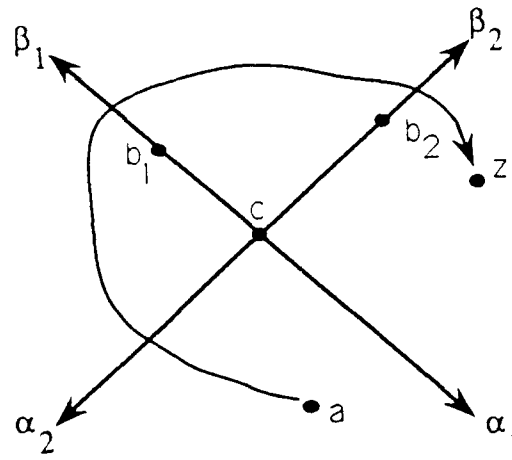


Figure 4.6 The Class $\alpha_2\beta_1\beta_2$

The cone of directions in which the first segment of p^* must lie is determined by considering the first element of the class name. Therefore, the first cone must be in a direction which crosses the α_2 reference ray.

We draw a line from the starting point a to b_2 and a second line from " a " parallel to α_2 and in the same direction as α_2 (Figure 4.7). This forms the cone of directions in which the first polygonal line segment of the shortest representative must lie.

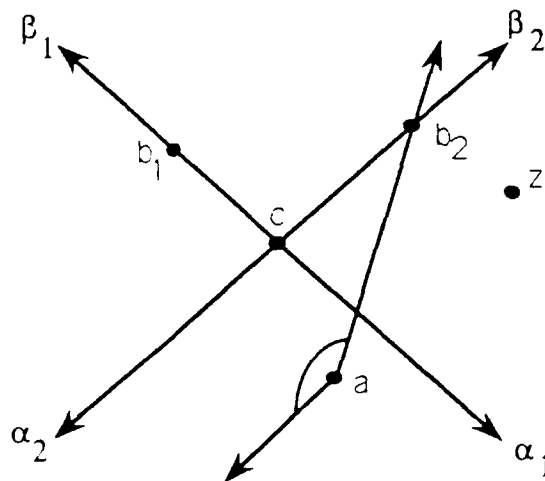


Figure 4.7 Cone of Directions Toward α_2

Then we consider the second character β_1 and create the cone of directions that will insure crossing of β_1 . We intersect the two cones of direction. The cone of intersections is pictured in Figure 4.8.

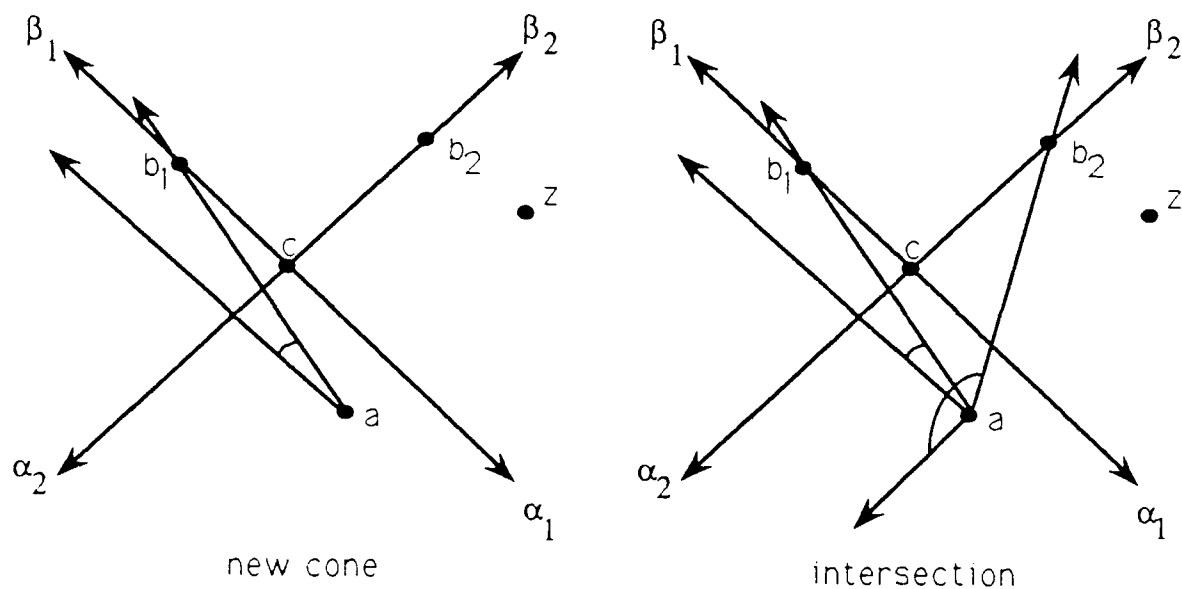


Figure 4.8 Two Intersecting Cones of Direction

Continuing in this fashion, in Figure 4.9, we consider the third character in the class, β_2 and create the appropriate cone.

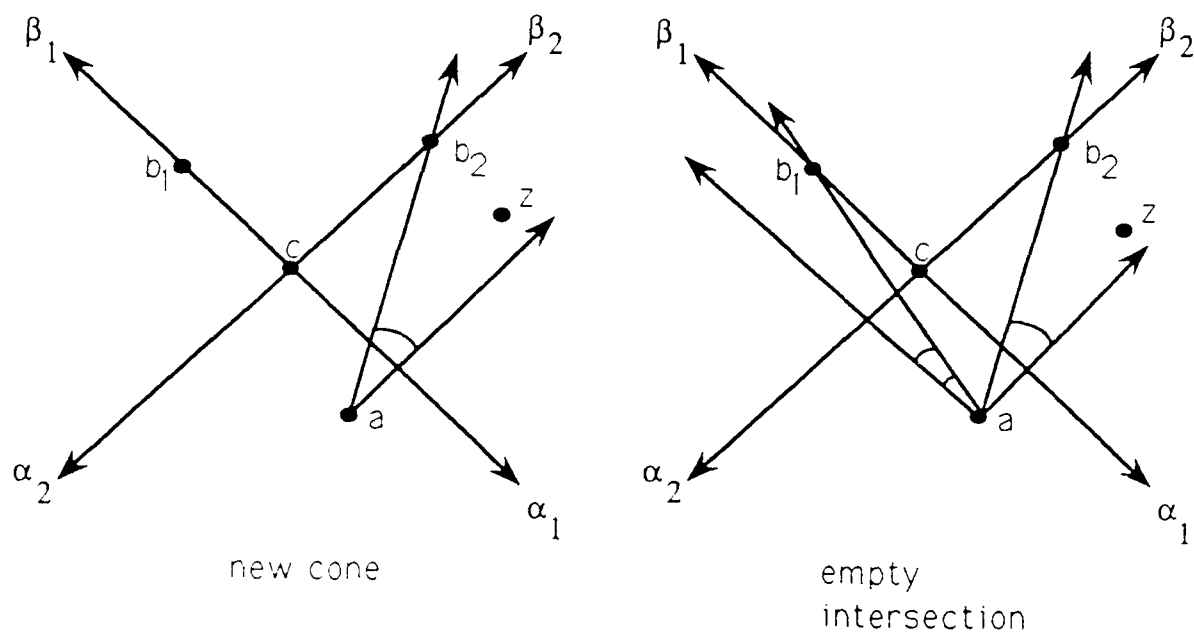


Figure 4.9 An Empty Intersection of Directions

We see that the intersection of these cones is empty. To enter this new cone we are forced out of the cone in which we know the first segment must lie. Therefore, we choose a path which crosses all rays that were named (α_2 and β_1) and turns on some b_k . We determine the correct b_k below.

4. An Empty Intersection

As can be seen in Figure 4.9, the new cone of directions lies to the right of the previous intersected cone of directions. This implies that the path we require will bend to the right. The peg--point b_k on which the path bends--is found on the right hand boundary of the old cone. In fact, as shown in Figure 4.9, the obstacle centered at b_1 is the peg that the path bends on in the class $\alpha_2\beta_1\beta_2$.

We now know that one segment of the bounding path is the line from a to b_1 . In the next step we use b_1 as the new starting point. Since α_2 and β_1 have been crossed, we use β_2 to create the cone of directions in Figure 4.10.

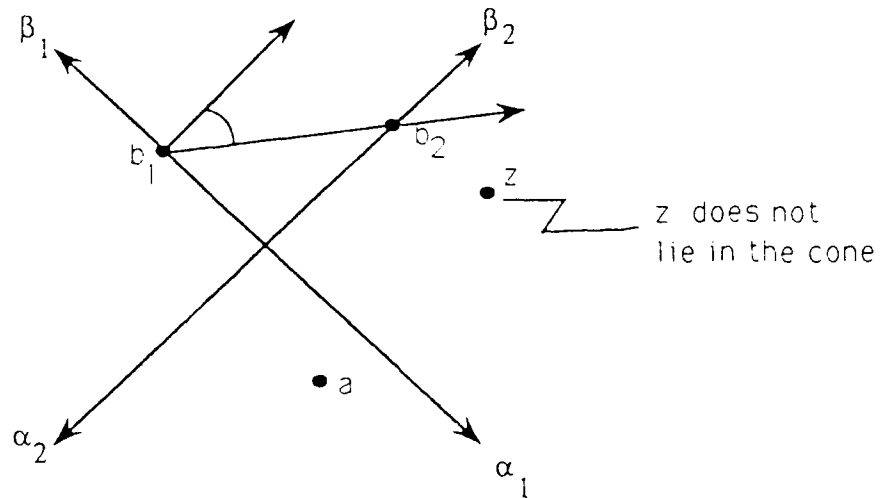


Figure 4.10 Cone of Directions

Now, since there are no more characters to read, we check to see if z lies within the most recently established cone of directions. In Figure 4.10 we see that z does not lie in the cone so the bounding path must bend again.

Since z lies to the right of the cone, the bounding path bends on b_2 which is on the right hand boundary of the cone of directions. Hence, the final two segments of the bounding path are the line segments between b_1 and b_2 and the line segment between b_2 and z .

So given the class $\alpha_2\beta_1\beta_2$ we obtain the bounding path pictured in Figure 4.11 on the contracted obstacles. Computing the length of this path is now straight forward.

The next section generalizes these steps and shows how to find the path to z .

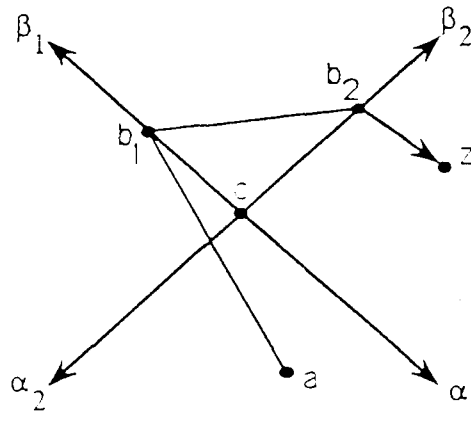


Figure 4.11 The Bounding Path for the Class $\alpha_1\beta_1\beta_2$

C. THE ALGORITHM

To find a lower bound for a path in a given class, the following algorithm is applied:

$t(j) :=$ an array of points with $t(1)=a$, and $t(m)=z$;
 $t(i+1)$ through $t(m-1)$ are pegs visited along the bounding polygonal path.
 $peg(i) :=$ the obstacle-representing point associated with x_i
 $cone(t(j), x_i) :=$ the cone of directions from a starting point $t(j)$ in which we can cross the ray x_i with a straight line from t
 $U :=$ represents the cone associated with the most recently read character
 $V_{old} :=$ records the most recent cone of intersections
 $V :=$ represents the cone of directions created by one character; V is assigned to V_{old} and is replaced by the intersection of U and V_{old}
 $L, R :=$ represent pointers into the character string X and is associated with the obstacle on the left of right boundary of the cone of directions. If no boundary lies on a ray, then L or R is assigned zero
 $L_0, R_0 :=$ associated with the obstacle on the left of right boundary of the cone U that is intersected with V_{old} . If one ray of U lies inside V_{old} , then either L or R is replaced by L_0 or R_0 . If both rays of U lie inside V_{old} then L and R are replaced by L_0 and R_0 respectively.

```

BEGIN
  t(1) <-- a; i <-- 1
  procedure START
    read character xi
    initialize current cone V <-- cone(t, xi)
    IF peg(i) on left boundary of V THEN
      L <-- i; R <-- 0
    ELSE
      R <-- i; L <-- 0
    END IF
  procedure CONTINUE
    i <-- i+1; read character xi
    U <-- cone(t, xi)
    IF peg(xi) on left boundary of V THEN
      LU <-- i; RU <-- 0
    ELSE
      RU <-- i; LU <-- 0
    END IF
    Vold <-- V; Lold <-- L; Rold <-- R
    V <-- U ∩ Vold
    IF left boundary of U falls within Vold THEN; L <-- LU; END IF
    IF right boundary of U falls within Vold THEN; R <-- RU; END IF
  procedure BRANCH
    IF V is empty THEN
      IF U is to left of Vold THEN
        j <-- j + 1; t(j) <-- peg(Lold)
      ELSE ( U is to right of Vold )
        j <-- j + 1; t(j) <-- peg(Rold)
      END IF
      go to procedure START
    ELSE (V is not empty.)
      IF i < m (not all characters have been read) THEN Go to procedure CONTINUE.
      ELSE (There are no more characters to read.)
        IF z lies in current cone V THEN j <-- j + 1; t(j) <-- z; STOP
        ELSE
          IF U is to left of Vold THEN
            j <-- j + 1; t(j) <-- peg(Lold); j <-- j + 1; t(j) <-- z; STOP
          ELSE ( U is to right of Vold )
            j <-- j + 1; t(j) <-- peg(Rold); j <-- j + 1; t(j) <-- z; STOP
          END IF
        END IF
      END IF
    END IF
  END IF
END

```

V. FINDING THE SHORTEST PATH

A. INTRODUCTION

In order to find the true shortest path from a to z we consider the classes of paths on the ordered list of candidate paths. We proceed by removing the first class from the list. The true shortest path in the named class is found. If the length of this path is smaller than the bound associated with the class on the top of the remaining ordered list, the search is stopped. Otherwise, the first class is removed from the remaining list and the above procedure is repeated. This continues until the true shortest path is found.

To find the shortest path within a given class, we reverse the contraction applied to the obstacles performed earlier when we found the lower bound for the classes (see Figure 5.1). For each polygonal path, whose length provided a lower bound, we transform the segment into the true shortest path in that class. Accordingly, the process of calculating the shortest path in a class begins with the polygonal path used to lower bound it and a description of the obstacle geometries.

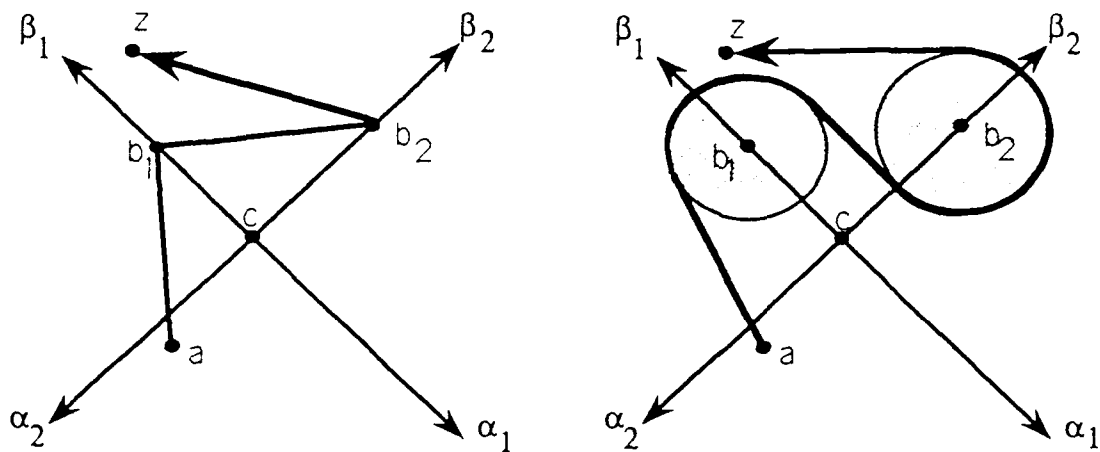


Figure 5.1 Reversing the Obstacle Contraction

To this point, no assumptions have been made about the shapes of the obstacles. In this chapter, with little loss of generality and for computational purposes which will be discussed later, all obstacles are assumed to be circles centered on b_k . We could, of course, assume other shapes, like polygons or ellipses.

B. FINDING LINE SEGMENTS

As can be seen in Figure 5.2 the line segment from a to B_1 is tangent to B_1 . Also, the line segment from B_1 to the obstacle B_2 is tangent to both obstacles. In general there are two tangent lines from a point to a circle and there are four distinct lines tangent between two circles (see Figure 5.2). In this section we see how to find these tangent lines and we show how to pick the correct tangent line to represent the true shortest path in the class.

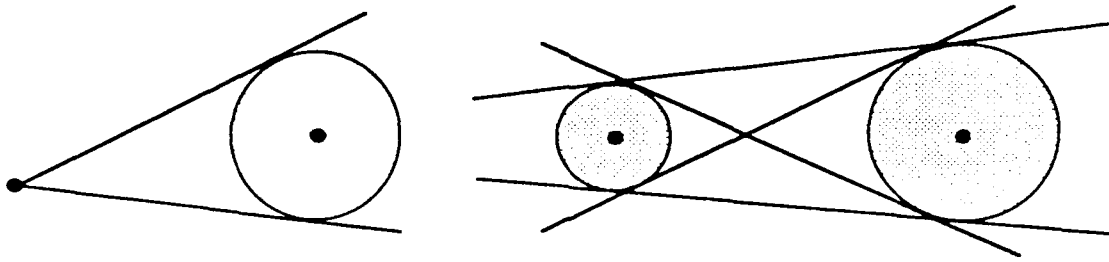


Figure 5.2 Possible Tangent Lines

Assuming that the shortest path from a to z in a given class of paths is not a straight line, we use another application of the cone of directions idea to determine on which side of an obstacle to turn. That is equivalent to picking the correct tangent line. We use the last cone of intersections formed just before the intersection of cones becomes empty to make this determination.

In Figure 5.3 we see that the path is required to bend around obstacle B_1 . If a tangent line is drawn from a to the left side of obstacle 1 and we then travel on an arc around the obstacle until we can proceed directly to z , we would have the shortest path in the class and could easily find its length. If not then we iterate the process.

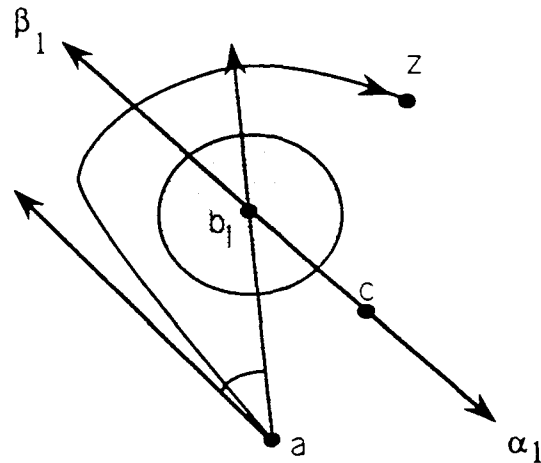


Figure 5.3 Path Bend in Class B_1

The following method is used to find the tangent lines to a circular obstacle when given the starting point, the coordinates of the center of the obstacle, and the radius of the obstacle.

In Figure 5.4, the starting point, a , is the point (u,v) . The center coordinates and the radius of the peg are the point (h,k) and r , respectively.

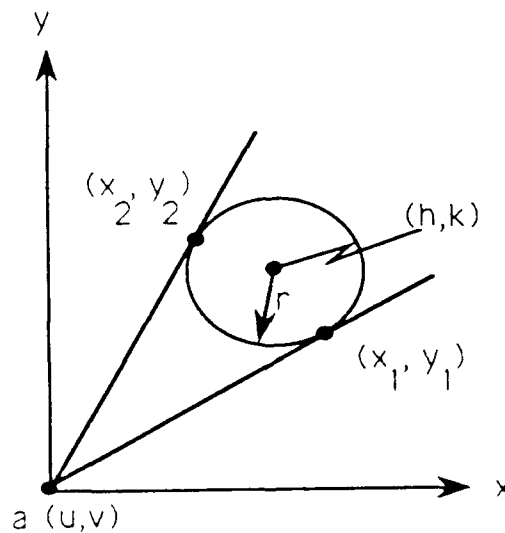


Figure 5.4 Point to Obstacle Tangent Lines

With this information we determine the equations of the circle representing the obstacle and an arbitrary line. We solve these equations simultaneously for the points of intersection (Figure 5.5). The specific value that we want is the point of tangency.

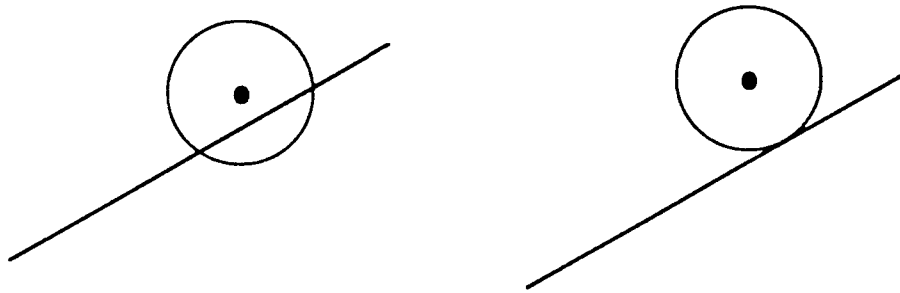


Figure 5.5 The Intersections of a Circle and a Line

So, given h , k , and r and assuming, without loss of generality, that $b=u=v=0$, we find the following two equations:

$$(x-h)^2 + (y-k)^2 = r^2$$

$$y = mx \text{ (where } m \text{ is the slope of the line)}$$

By expanding the first equation and substituting $y=mx$ we obtain:

$$(1+m^2)x^2 + (2mk-2h)x + (h^2+k^2-r^2) = 0 .$$

By the quadratic formula we obtain:

$$x = \frac{-2mk+2h \pm [(2mk-2h)^2 - 4(1+m^2)(h^2+k^2-r^2)]^{1/2}}{2(1+m^2)} \quad (1)$$

In Equation (1), if the discriminant is zero, we have found the value where the points of intersection coincide. So setting the discriminant equal to zero yields:

$$\begin{aligned}
(-2h-2km)^2 - 4(1+m^2)(h^2+k^2-r^2) &= 0 \\
(h+km)^2 - (m^2+1)(h^2+k^2-r^2) &= 0 \\
(k^2m^2+2hkm+h^2) - (m^2+1)(h^2+k^2-r^2) &= 0 \\
(k^2-h^2-k^2+r^2)m^2 + (2hk)m + (r^2-k^2) &= 0
\end{aligned}$$

from which m can be determined.

A second application of the quadratic formula produces:

$$\begin{aligned}
m &= \frac{-2hk \pm [4h^2k^2 - 4(r^2-h^2)(r^2-k^2)]^{1/2}}{2(r^2-h^2)} \\
m_i &= \frac{-hk \pm r[h^2+k^2-r^2]^{1/2}}{r^2-h^2}, \quad i=1,2.
\end{aligned}$$

In general, with an arbitrary u and v

$$m_i = \frac{-(h-u)(k-v) \pm r[(h-u)^2 + (k-v)^2 - r^2]^{1/2}}{r^2 - (h-u)^2}, \quad i=1,2$$

The only time this technique does not appear to work is when the tangent line found is a vertical line (m is undefined because there is a zero in the denominator). This problem can be easily handled because if m is undefined we know the equation for a vertical line to be x equal to some constant.

The value for m is then substituted into Equation (1) to find the x_i , $i = 1,2$, coordinate and the value of x_i is substituted into $y_i = m_i x_i$ to find y_i . By knowing on which side of the obstacle the path must turn, we can pick the tangent line with the appropriate slope. A method of picking the correct line will be discussed in the next section.

We follow the arc along the circle to the point of tangency of the line between the current obstacle and z or to the point of tangency to the next obstacle. Tangent lines between two obstacles can be determined by a similar method as the one just shown. A method of finding the four tangent lines between two circular obstacles follows.

C. DETERMINING THE CORRECT TANGENT LINES

1. Point to Obstacle

We have shown how to find the two tangent lines from a point to a circle. We present a method to determine which tangent line to pick.

The obstacle on which we turn lies on a boundary of the cone of directions. Thus the cone itself lies either to the left or to the right of the ray through the obstacle center. If the cone lies on the left, use the tangent of the left is taken. If the cone lies to the right, then the tangent on the right is taken. The case with the cone on the left is illustrated in Figure 5.6.

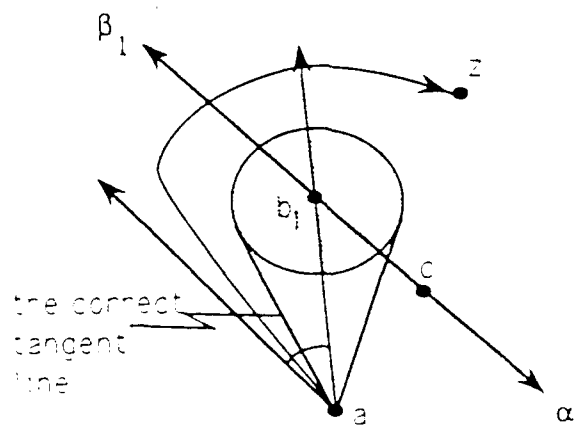


Figure 5.6 Use the Left Tangent for Path Bending to the Right

2. Obstacle to Obstacle

When considering two circular obstacles, there are four distinct lines that can be drawn tangent to both obstacles. In this section we show how to pick the tangent line that corresponds to the respective shortest path in the given class.

To find the four distinct tangent lines between two obstacles we first create a line which is tangent to one of the circles. We then move this tangent line along the circle boundary until we discover the four points at which the line is tangent to both obstacles (Figure 5.7).

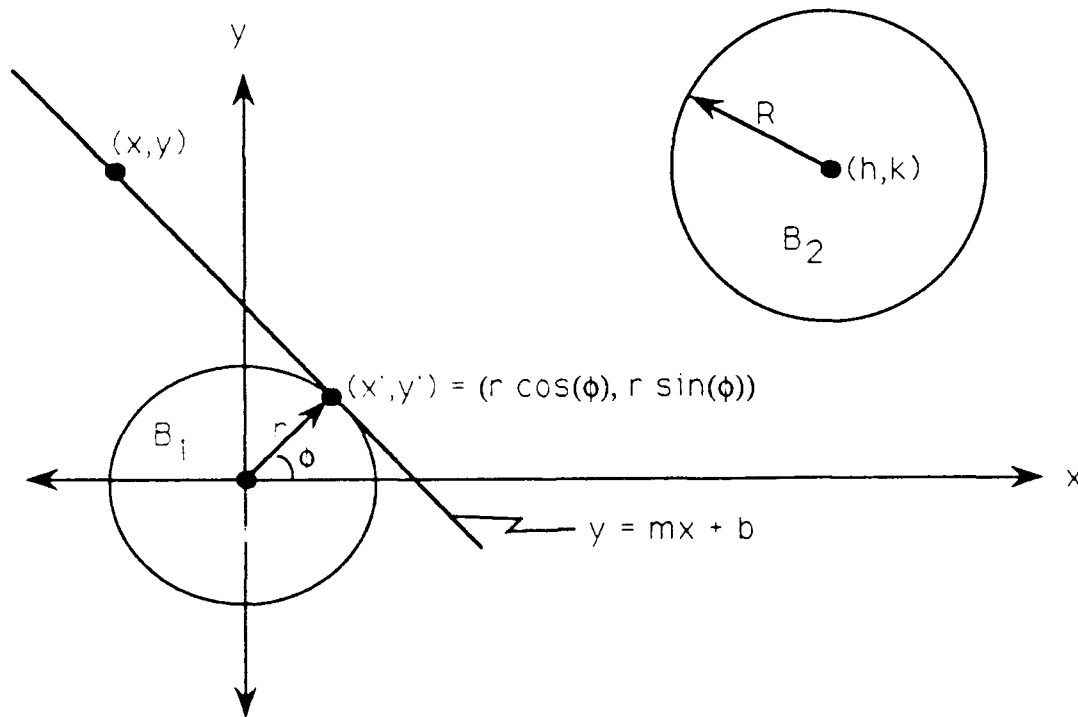


Figure 5.7 Rotating ϕ to Find Tangent Lines Between Two Circles

Without loss of generality, in Figure 5.7 we fix one obstacle center at the origin. We pick an arbitrary point (x', y') , on the obstacle B_1 represented as $(r \cdot \cos(\phi), r \cdot \sin(\phi))$. We construct a line through (x', y') tangent to B_1 . The slope m of this line is:

$$\frac{y - r \cdot \sin(\phi)}{x - r \cdot \cos(\phi)} .$$

This slope is also equal to $-\cot(\phi)$. Hence,

$$m = \frac{y - r \cdot \sin \phi}{x - r \cdot \cos \phi} = -\cot(\phi) \quad (2)$$

The equation of the circle with center (h, k) is

$$(x-h)^2 + (y-k)^2 = R^2$$

Rearranging Equation (2) we obtain the following:

$$\begin{aligned} y - r \cdot \sin(\phi) &= (-\cot(\phi)) (x - r \cdot \cos(\phi)) \\ y &= (-\cot(\phi))x + r \cdot \cos(\phi) \cdot \cot(\phi) + r \cdot \sin(\phi) \end{aligned} \quad (3)$$

Equation (3) is in slope-intercept form and the y intercept b is:

$$\begin{aligned}
b &= r \cos(\phi) \cot(\phi) + r \sin(\phi) \\
&= r \left(\frac{\cos^2 \phi}{\sin \phi} + \sin \phi \right) \\
&= r \left(\frac{\cos^2 \phi + \sin^2 \phi}{\sin \phi} \right) \\
b &= r \csc(\phi)
\end{aligned}$$

Thus both m and b are functions of ϕ . The general line that we use to intersect both circles is

$$y = m(\phi)x + b(\phi) .$$

This equation for the line and for the obstacle centered at (h,k) are solved simultaneously. After steps similar to those used in the point to obstacle example we obtain the following formula:

$$(\pi^2+1)x^2 + [-2h+2m(b-k)]x + [h^2+(b-k)^2-R^2] = 0 .$$

Again by applying the quadratic formula and setting the discriminant equal to zero we find

$$4[-h-m(b-k)]^2 - 4(\pi^2+1)[h^2+(b-k)^2-R^2] = 0 .$$

Substituting the function values for $m(\phi)$ and $b(\phi)$ yields

$$[-h-(\cot \phi)(r \csc \phi - k)]^2 - (\cot^2 \phi + 1)[h^2 + (r \csc \phi - k)^2 - R^2] = 0 \quad (4)$$

This equation has four zeros for values of ϕ in $[0, 2\pi)$ as can be seen in Figure 5.8.

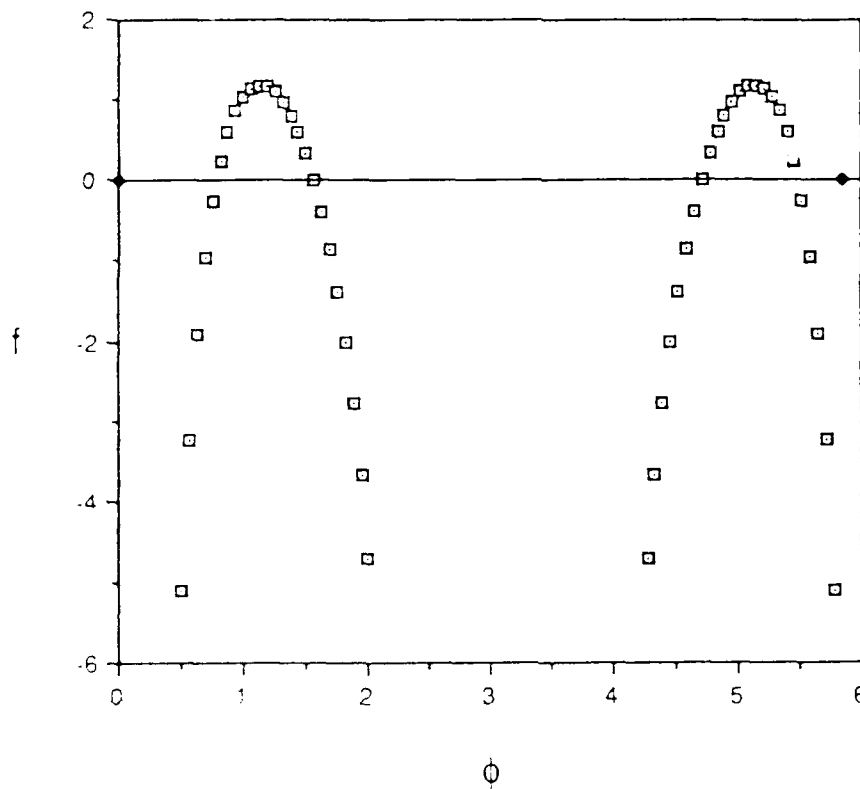


Figure 5.8 Four Zeros of Equation (4)

Since ϕ is the only parameter in Equation (4) we can use the four values found to label the four tangent lines. The line associated with the lowest value of ϕ is labeled A, the second line associated with the second lowest ϕ value is labeled B and the last two lines are labeled with C and D, respectively.

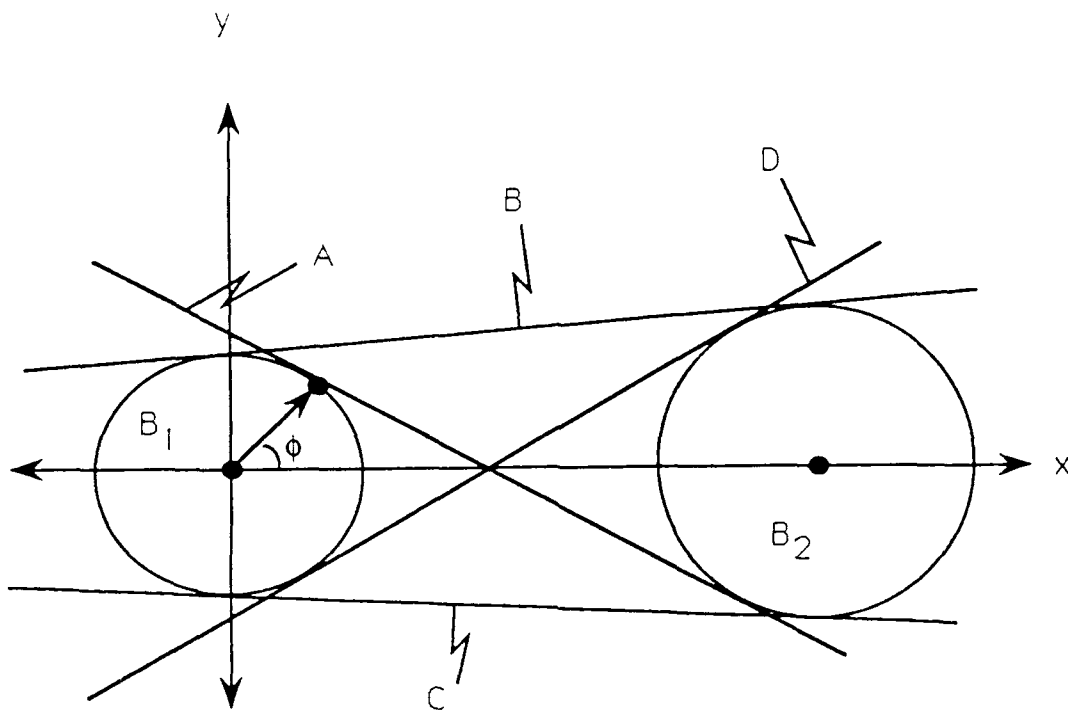


Figure 5.9 Possible Tangent Lines

Assuming the polygonal path bends on each peg on the path, there are essentially two possible arrangements when considering a polygonal path that negotiates two obstacles. These possibilities are shown in Figure 5.10. The paths can be flipped about the horizontal axis to produce two equivalent paths.

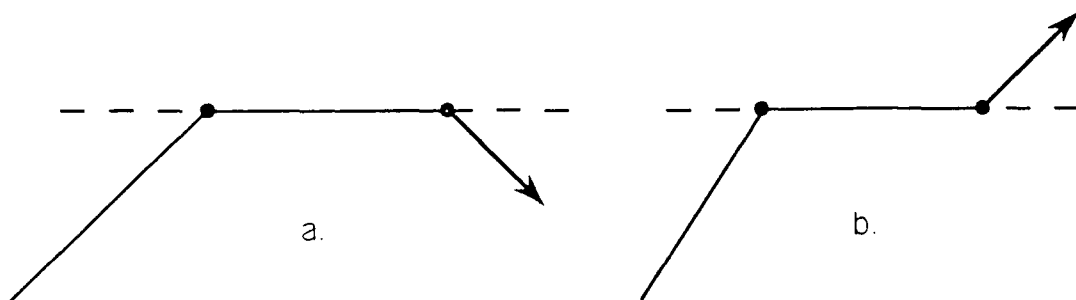


Figure 5.10 Two Possible Polygonal Paths

In order to discover which situation exists, we extend the middle line segment and use it as a reference in Figure 5.10. Since the path bends at each peg, the two end segments must lie on one side or the other of the reference line. If they both lie on the same side, then we have situation a above. If the two segments lie on opposite sides, then we have situation b.

Given the two possible arrangements in Figure 5.10, we know the corresponding shortest paths in the class look like those in Figure 5.11.

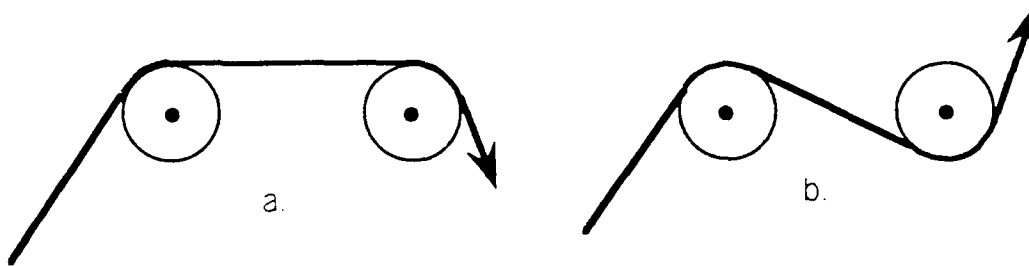


Figure 5.11 True Shortest Paths

We have shown how to pick the correct tangent line from a point to a circle. This is applied to find the first and last segments of this polygonal path.

To find the correct tangent between two obstacles, following the situation on the left in Figure 5.11 we must pick tangent line B, described earlier. If we have the other situation in Figure 5.11, we must pick tangent line A.

If the polygonal path does not bend on each peg, then there are two possible configurations as seen in Figure 5.12.

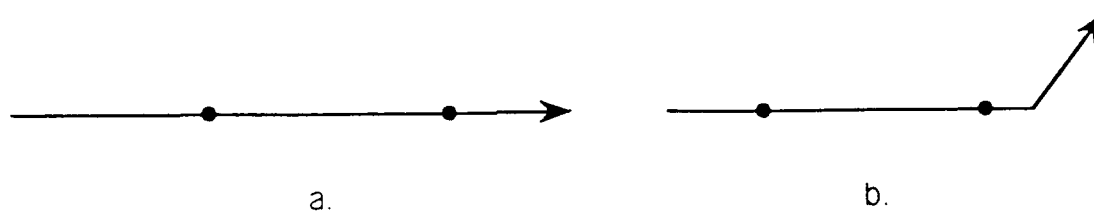


Figure 5.12 Polygonal Paths With Collinear Segments

To this point, we only used the cone of directions idea to find the correct tangent from a point to an obstacle. We use the cone of directions information to pick the correct tangent lines too.

As seen in Figure 5.13 we consider the class $B_1\alpha_2$.

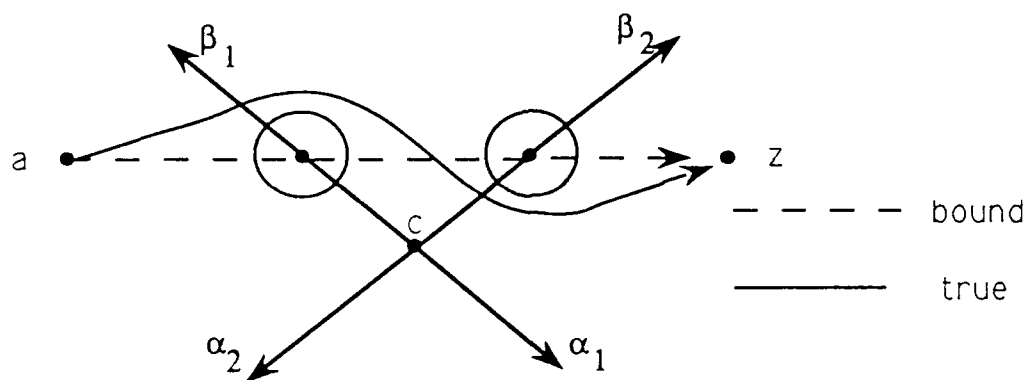


Figure 5.13 Class $B_1\alpha_2$ and its Bounds

The tangent line from a to B_1 is found in the same manner as discussed earlier. However, since the path of the bound does not bend, we cannot choose the correct tangent line between B_1 and B_2 in the same way as before.

Instead, we construct a reference line that connects the centers of the two obstacles. Then we consider on which side of this line the cones of directions lie (Figure 5.14).

If they lie on different sides of the reference line then we pick tangent line A or D. Otherwise, we pick tangent lines B or C.

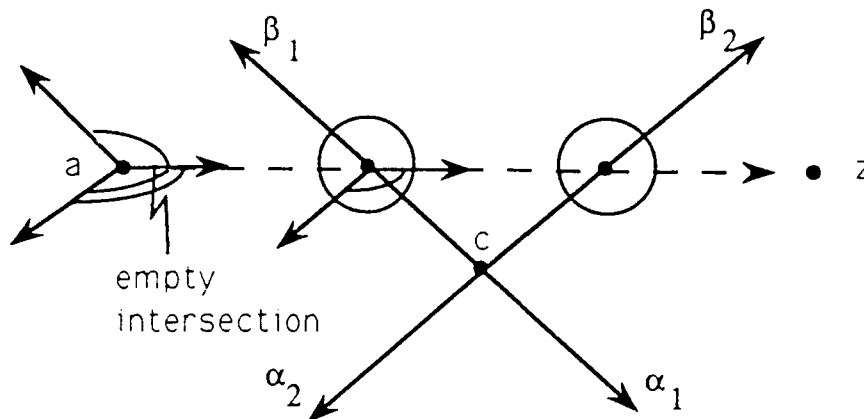


Figure 5.14 Cone of Direction to Help Pick Tangent Lines

D. TWO POTENTIAL PROBLEMS TO CHECK

Given a path in a class like the one in Figure 5.15, we see the associated bounding path when we shrink the obstacles to points. However, there are two separate problems that may arise when the obstacles are restored to their original shape:

- 1) The prospective path may pass through one or more obstacles.
- 2) When the obstacles are restored to their full size the path created can switch sides of one or more obstacles.

These possibilities can be seen in Figure 5.16.

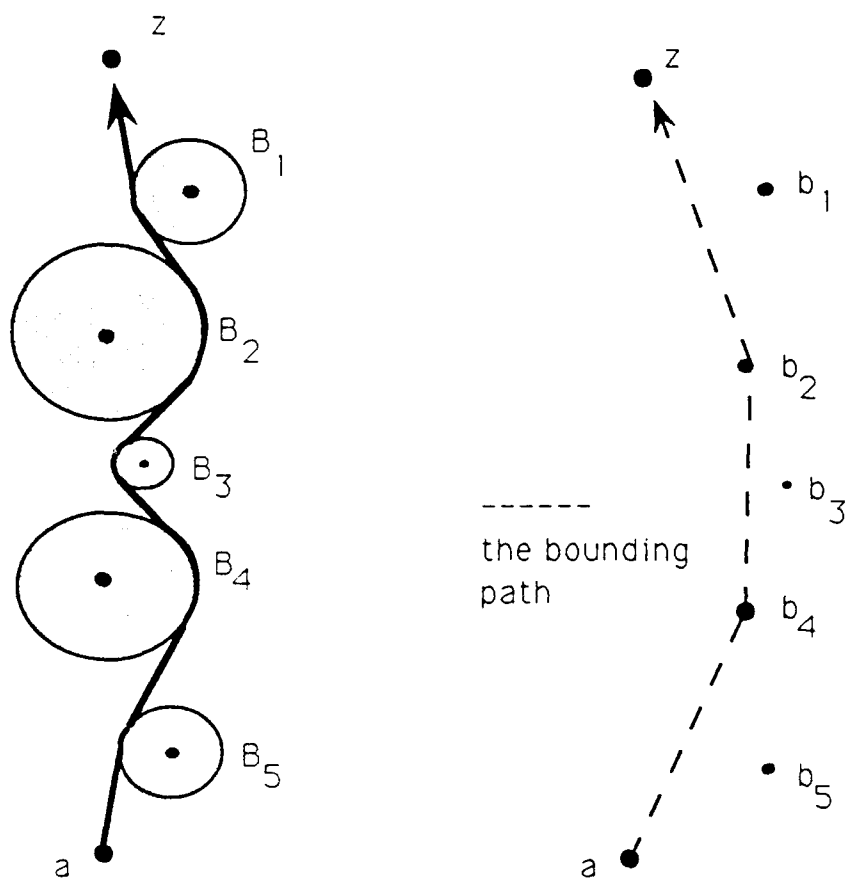


Figure 5.15 A Class and Its Bounding Path

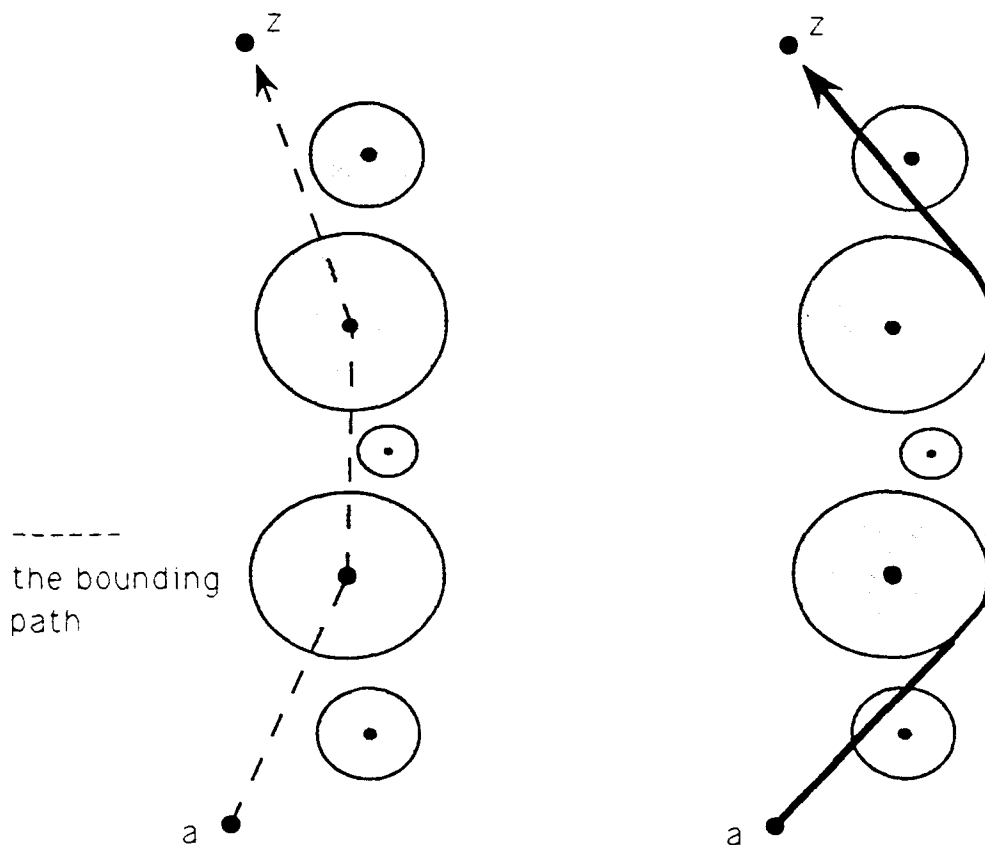


Figure 5.16 A Bounding Path and Its Restored Path

The path determined by the cone of directions technique passes through obstacles B_1 and B_5 , illustrating the first problem. By switching sides of B_3 , the path is no longer in the same equivalence class, illustrating the second problem. Actually, the path should look like the one in Figure 5.17 which avoids all obstacles and remains in the original class.

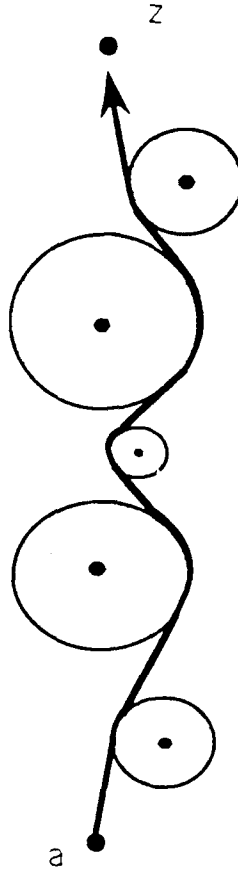


Figure 5.17 Corrected Shortest Path

The correct path around the obstacles can be determined by using a distance function. Then, methods discussed in Chapter IV are employed to negotiate the new impeding obstacles. In Figure 5.18, segment P_j^* --which is one segment of the bounding path--joins pegs b_k and b_{k+1} , and lies along the line through these pegs. We let a circular obstacle B_j have center (u_j, v_j)

and radius r_j . Pegs b_k and b_{k+1} have cartesian coordinates (x_i, y_i) and (x_{i+1}, y_{i+1}) respectively. We let $d(*,*)$ denote Euclidean distance.

The distance from p_s^* to B_j --which is the perpendicular distance from p_s^* to the nearest point in B_j --is given by $d(p_s^*, B_j)$ if the center of B_j lies in the band denoted in Figure 5.18 whose borders are perpendicular to p_s^* and which passes through b_k and b_{k+1} , and by $\min\{d(b_k, B_j), d(b_{k+1}, B_j)\}$ if not. We can then check whether the center of B_j is in the band if we define vectors

$$\lambda_i = ([x_{i+1} - x_i], [y_{i+1} - y_i])$$

and

$$\mu_{ij} = ([u_j - x_i], [v_j - y_i])$$

which can be seen in Figure 5.19 [Ref. 5].

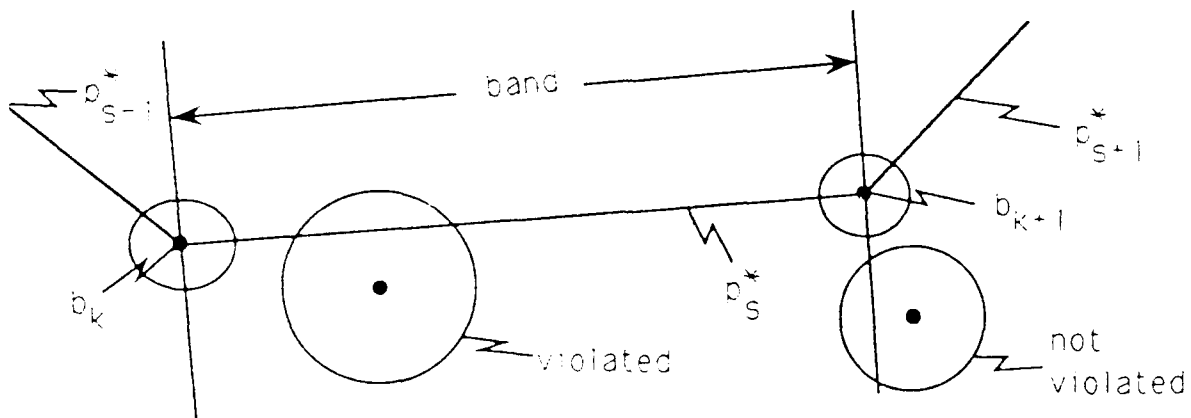


Figure 5.18 Band Created By One Segment of a Polygonal Path

Then the center of B_j is in the band if and only if $\lambda_i \cdot \mu_{i-1,j} > 0$ and $\lambda_i \cdot \mu_{i,j} < 0$, where (\cdot) denotes the inner product. We write the distance function corresponding to p_s^* passing through B_j as

$$d(p_s^*, B_j) = \begin{cases} d(p_s^*, b_j); & \text{if } \lambda_i \cdot \mu_{i-1,j} > 0 \text{ and } \lambda_i \cdot \mu_{i,j} < 0 \\ \min d(b_k, B_j), d(b_{k+1}, B_j); & \text{otherwise} \end{cases}$$

These distances are easily computed with

$$d(b_{k+1}, B_j) = [(x_i - u_j)^2 + (y_i - v_j)^2]^{1/2} - r_j$$

and

$$d(p_s^*, B_j) = \frac{|(y_{i+1} - y_i)y_i - (x_{i+1} - x_i)v_j - x_i(u_{i+1} - h_i) + y_i(x_{i+1} - x_i)|}{[(y_{i+1} - y_i)^2 + (x_{i+1} - x_i)^2]^{1/2}} - r_j$$

[Ref. 5].

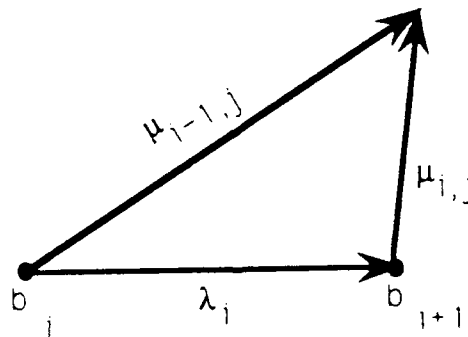


Figure 5.19 Vectors to Help Determine $d(p_s^*, B_j)$

With this new information, appropriate tangent lines must be found in order to avoid newly-violated obstacles and to keep the path in the correct class. These new tangent lines may also encroach obstacles. So the process of checking all obstacles is continued until no obstacles are violated.

To find the length of the shortest path, we sum the straight line segments and all of the arc lengths corresponding to the distances traveled around each circular obstacle.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

Determination of the shortest path between two points in a plane containing obstacles has applications in many fields, particularly that of robotic path planning. This thesis addresses several key issues belonging to a larger plan for the solution of the shortest path problem. This plan may be summarized as follows:

1. Partition the set of paths into homotopy classes.
2. Associate algebraic "names" with each of these classes.
3. Produce a finite list of feasible classes.
4. For each class on the above list, calculate an inexpensive lower bound on the length of its shortest path.
5. Arrange the list in increasing order of these lower bounds.
6. Considering the classes in listed order, calculate the shortest path in the class. Calculation terminates when some class is encountered whose shortest path has length shorter than the lower bound associated with the succeeding class.

The approach outlined above offers two primary computational advantages. First, note that the collection of possible paths from a given point of origin to a given destination is uncountably infinite. By organizing the search around homotopy classes and by taking advantage of certain topological relationships in the region [Ref. 3], the search of an uncountably large set is replaced by a search of a finite list.

Second, the search of this finite list is ordered according to lower bounds which estimate the lengths of the associated shortest paths. This ordering allows the search to terminate without exhausting the list. The bounds associated with each class are inexpensive to calculate--relative to the cost of finding the class' shortest path. This results in further savings of computational effort.

The contributions of this thesis concentrate in areas 2, 4, and 6 of the above list.

As regards the association of names with homotopy classes (area 2) we present a computational investigation which was conducted ancillary to proof activity in this area. In summary, we associate a name with each homotopy class as follows: by imposing a certain labeled reference frame on the region in question; by encoding information about the relationship of a given path with the frame; and by employing algorithms presented here. In this way, we reduce the encoded information to a string of characters which is unique to each class. The investigation presented here uses a presentation of the well-known fundamental group as a standard of comparison to verify that the character strings used in our approach are in fact class names. (This proposition has subsequently been proved [Ref. 2]). The significance of the structure of the names which we employ is that they incorporate information which is useful both for the

calculation of class bounds and for calculation of class shortest paths.

For the calculation of class bounds (area 4) we represent each obstacle by a point chosen within it. This point is used to construct the reference frame which is imposed on the region. Restricting our attention to such summary information allows class bounds to be calculated cheaply.

The final step (area 6) begins with an ordered list of classes. In this paper we present a method by which shortest paths of a class may be calculated when the obstacles are circular. This is the only step in which any generality is lost, and is potentially fruitful subject for further study.

B. RECOMMENDATIONS FOR FURTHER STUDY

The field of robotics is one that is gaining much interest in mathematics and other fields of study. Further analysis of the reference frame and its properties may aid future researchers in answering other questions concerning the shortest path problem.

One aspect of the problem that can be studied further relates to areas 4 and 6 above. Programming the two steps of finding the lower bound for a class and subsequently determining the shortest path in a class would be quite useful.

Another computational problem relates to area 2. It is believed that further study and refinement of the code could result in a program which is even more efficient.

Another possible subject of future research is that of considering non-circular obstacles (area 6). It should be noted that any set of obstacles can be covered with circles. With this approximation the procedures presented in this thesis may be applied.

Finally, this thesis introduces only part of the solution to the shortest path problem. The thesis by CAPT Kevin D. Jenkins, U.S. Marine Corps [Ref. 3], presents the remaining portions of the solution (area 3), and should be studied in conjunction with this paper.

APPENDIX A. FORTRAN PROGRAM: THE COMPUTATIONAL INVESTIGATION

PROGRAM CLASNAME FORTRAN

```

C *****
C THIS PROGRAM RANDOMLY GENERATES OBSTACLES AND A POLYGONAL PATH
C THROUGH THOSE OBSTACLES TO DETERMINE WHETHER OR NOT THE PROCEDURES
C USED BY COMPETING ALGORITHMS PRODUCE THE SAME FUNDAMENTAL GROUP
C REPRESENTATION FOR THAT PATH.
C *****

C THIS PORTION OF THE PROGRAM SERVES AS THE MAIN DRIVER WHICH RECEIVES
C THE PARAMETERS DEFINING THE REGION AND INITIALIZES THE LINK LIST
C ARRAYS WHICH WILL BE USED TO REPRESENT THE POLYGONAL PATH.

C INPUT:  INITIAL SEED FOR THE RANDOM NUMBER GENERATOR, NUMBER OF
C          OBSTACLE CONFIGURATIONS AND PATHS TO BE TESTED, NUMBER OF
C          OBSTACLES IN THE PLANE AND NUMBER OF SEGMENTS IN EACH
C          POLYGONAL PATH

C OUTPUT: FINAL RANDOM NUMBER GENERATOR SEED AND MESSAGES INDICATING
C          ANY PATHS WHICH PRODUCE DIFFERENT FUNDAMENTAL GROUP
C          REPRESENTATIONS

      REAL*8  BX(1000), BY(1000), X(1000), Y(1000), DSEED
      INTEGER NOBS, NSEGS, NUMPTS, N, M, HEAD(1000)
+      NEXT(1000), PRED(1000), BOARD, PATH
      EXTERNAL GGUBFS

      DSEED = 123457.0
      i; = 1000
      N = 1000
      NOBS = 20
      NSEGS = 5
      NUMPTS = NSEGS + 1
      PRINT*, 'INPUT SEED', DSEED

      DO 1 BOARD = 1, N
        CALL BOARDS (DSEED, BX, BY, NOBS)
        DO 2 PATH = 1, M
          CALL INIT (HEAD, NEXT, PRED, PSEED, DSEED )
          CALL PATHS (DSEED, X, Y, NSEGS)
          CALL TEST(HEAD, NEXT, PRED, BX, BY, NOBS, X, Y, NUMPTS, BSEED, PSEED)
2        CONTINUE
1      CONTINUE

      PRINT*, ' '
      PRINT*, 'FINAL SEED', DSEED

      STOP
      END

```

```

      SUBROUTINE INIT ( HEAD, NEXT, PRED , PSEED, DSEED)
C *****

C THIS SUBROUTINE INITIALIZES THE DOUBLE LINK LIST ARRAYS THAT WILL
C REPRESENT THE PATH.

C INPUT: DSEED

C OUTPUT: HEAD, NEXT, AND PRED ARRAYS SET TO ZERO, AND SEED FOR THE
C R.N.G. PRIOR TO CONSTRUCTION OF THE PATH

      INTEGER HEAD(1000), NEXT(1000), PRED(1000)
      REAL*8 PSEED, DSEED

      PSEED = DSEED

      DO 3 K = 1,1000
        HEAD(K) = 0
        NEXT(K) = 0
        PRED(K) = 0
      3 CONTINUE

      RETURN
      END

```

```

      SUBROUTINE BOARDS (DSEED, BX, BY, NOBS)
C *****

C THIS SUBROUTINE USES A PSEUDO RANDOM NUMBER GENERATOR TO CREATE THE
C COORDINATES OF EACH OBSTACLE ON THE BOARD, SCALING ALL COORDINATES
C TO BE IN THE INTERVAL (-1,1).

C INPUT: NUMBER OF OBSTACLES IN THE REGION AND A SEED FOR THE R.N.G.

C OUTPUT: OBSTACLE COORDINATES

      REAL*8 BX(1000), BY(1000), DSEED
      EXTERNAL GGUBFS

      DO 1 I = 1, NOBS
        BX(I) = 2. * GGUBFS(DSEED) - 1.
1      BY(I) = 2. * GGUBFS(DSEED) - 1.

      RETURN
      END

```

```

      SUBROUTINE PATHS (DSEED, X, Y, NSEGS)
C *****

C  RANDOMLY GENERATES THE X AND Y COORDINATES OF THE VERTICES FOR THE
C  POLYGONAL PATH, SCALING ALL COORDINATES TO BE IN THE INTERVAL (-1,1).
C  THIS SUBROUTINE ALSO ENSURES THAT THE PATH IS A CLOSED LOOP BY
C  ASSIGNING THE START/FINISH POINTS THE SAME COORDINATES.

C  INPUT:  NUMBER OF PATH SEGMENTS AND A SEED FOR THE R.N.G.

C  OUTPUT: COORDINATES OF VERTICES ALONG THE POLYGONAL PATH

      REAL*8 X(1000), Y(1000), DSEED
      EXTERNAL GGUBFS

      DO 1 I = 1, NSEGS
        X(I) = 2. * GGUBFS(DSEED) - 1.
1       Y(I) = 2. * GGUBFS(DSEED) - 1.

      X(NSEGS+1) = X(1)
      Y(NSEGS+1) = Y(1)

      RETURN
      END

      SUBROUTINE TEST(HEAD, NEXT, PRED, BX, BY, NOBS, X, Y, NUMPTS,
+                   BSEED, PSEED)
C *****

C  GENERATES THE RAW STRING OF CHARACTERS REPRESENTING A PATH AND
C  DETERMINES ITS FUNDAMENTAL GROUP REPRESENTATION USING ALGORITHMS 1
C  AND 2. RESULTS OF THESE COMPETING ALGORITHMS ARE THEN COMPARED FOR
C  DIFFERENCES.

C  INPUT:  A POLYGONAL PATH AND COORDINATES OF POINTS REPRESENTING
C          OBSTACLES

C  OUTPUT: COMPARED RESULTS OF ALGORITHMS

      INTEGER HEAD(1000), NEXT(1000), PRED(1000), FR(1000), NR(1000),
+          PR(1000), FL(1000), NL(1000), PL(1000)
      REAL*8  BX(1000), BY(1000), X(1000), Y(1000)

      DATA FR/1000*0/
      DATA NR/1000*0/
      DATA PR/1000*0/
      DATA FL/1000*0/
      DATA NL/1000*0/
      DATA PL/1000*0/

```

```

CALL RAWSTR(BX, BY, NOBS, X, Y, NUMPTS, HEAD, NEXT, PRED, NELEMS)
CALL ALG2 (NOBS, NELEMS, HEAD, NEXT, X, Y, BX, BY, FR, NR, PR)
CALL CANALG2 (FR, NR, PR)
CALL ALG1 (HEAD, NEXT, PRED, NELEMS)
CALL ALG2 (NOBS, NELEMS, HEAD, NEXT, X, Y, BX, BY, FL, NL, PL)
CALL CTREX (FL, NL, PL, FR, NR, PR, BSEED, PSEED)

```

```

RETURN
END

```

```

      SUBROUTINE RAWSTR(BX, BY, NOBS, X, Y, NUMPTS, HEAD, NEXT, PRED,
+                      NELEMS)
C *****

```

```

C PRODUCES THE RAW STRING OF CHARACTERS WHICH REPRESENTS THE PATH,
C PAYING CLOSE ATTENTION TO THE ORDERING OF THE CHARACTERS WHERE
C NECESSARY. THE STRING IS CONSTRUCTED BY IDENTIFYING THOSE OBSTACLE
C REFERENCE LINES WHICH ARE CROSSED AS EACH PATH SEGMENT IS TRAVERSED
C IN ORDER.

```

```

C INPUT: ORDERED LIST OF VERTICES REPRESENTING THE PATH AND A POINT
C         REPRESENTING EACH OBSTACLE IN THE REGION

```

```

C OUTPUT: A RAW STRING OF ALPHAS AND BETAS CONTAINED IN AN ARRAY
C          NAMED HEAD() AND ITS PARALLEL ARRAYS NEXT() AND PRED() WHICH
C          PRODUCE THE DOUBLE LINKED LIST

```

```

      IMPLICIT REAL*8 (A-H,O-Z)
      INTEGER HEAD, NEXT, SEG, FSTSTR, HDINDX, SEGEND, FIRST, START,
+          LENGTH, PRED
      LOGICAL ALLALF
      DIMENSION X(1000), Y(1000), BX(1000), BY(1000), A1(1000),
+          B1(1000), A2(1000), B2(1000), D2(1000), DIST(1000),
+          FIRST(1000), HEAD(1000), NEXT(1000), PRED(1000)

```

```

      DATA DIST/1000*0.0/
      DATA FIRST/1000*0/

```

```

      CALL SETUP(NUMPTS,NOBS,CX,CY,X,Y,BX,BY,A1,B1,A2,B2,D2)

```

```

      HDINDX = 2
      NSEGS = NUMPTS - 1
      LOLD = 1
      HEAD(LOLD) = 0

```



```

DO 7 SEG = 1, NSEGS
  SEGEND = SEG + 1
  FSTSTR = HDINDX
  LSTSTR = HDINDX
  ALLALF = .TRUE.
  LENGTH = 0

  DO 6 LINE = 1, NOBS
    CHECK1 = (A1(LINE)*X(SEG)+B1(LINE)*Y(SEG))*
+      (A1(LINE)*X(SEGEND)+B1(LINE)*Y(SEGEND))
    IF (CHECK1.LT.0) THEN
      LENGTH = LENGTH + 1
      CHECK2 = (A2(SEG)*BX(LINE)+B2(SEG)*
+      BY(LINE)+D2(SEG))*D2(SEG)
      IF (CHECK2.LT.0) THEN
        HEAD(HDINDX) = -LINE
        LSTSTR = HDINDX
        HDINDX = HDINDX + 1
      ELSE
        CALL CASES1 (A1, B1, A2, B2, D2, SEG, LINE, XINT,
+      YINT)
        DISTC = XINT**2 + YINT**2
        DISTB = (XINT-BX(LINE))**2 + (YINT-BY(LINE))**2
        IF (DISTB.LT.DISTC) THEN
          HEAD(HDINDX) = LINE
          LSTSTR = HDINDX
          HDINDX = HDINDX + 1
          ALLALF = .FALSE.
        ELSE
          HEAD(HDINDX) = -LINE
          LSTSTR = HDINDX
          HDINDX = HDINDX + 1
        ENDIF
      ENDIF
    ENDIF
  ENDIF
6 CONTINUE
  IF (LENGTH.NE.0) THEN
    IF (ALLALF) THEN
      CALL ALPHAS (NEXT, LOLD, FSTSTR, LSTSTR, SEG, START,
+      HEAD, NSEGS, PRED)
    ELSE
      CALL ORDER(SEG, LINE, HDINDX, HEAD, NEXT, A1, A2,
+      B1, B2, D2, FSTSTR, LSTSTR, X, Y, START,
+      LENGTH, FIRST, LOLD, NSEGS, PRED)
    ENDIF
  ENDIF
7 CONTINUE

CALL COUNTR(START, NEXT, HEAD, NELEMS)
RETURN
END

```

```

      SUBROUTINE SETUP(NUMPTS, NOBS, CX, CY, X, Y, BX, BY, A1, B1, A2,
+      B2, D2)
C *****

C FOR EACH OBSTACLE, THIS ROUTINE DETERMINES THE COEFFICIENTS OF THE
C EQUATION FOR THE REFERENCE LINE FROM THE OBSTACLE TO THE ORIGIN. IN
C ADDITION, IT CALCULATES THE COEFFICIENTS OF THE LINE REPRESENTING
C EACH SEGMENT OF THE POLYGONAL PATH.

C INPUT:  NUMPTS, NOBS, COORDINATES OF VERTICES ALONG POLYGONAL PATH
C         AND COORDINATES OF THOSE POINTS WHICH REPRESENT EACH
C         OBSTACLE

C OUTPUT: COEFFICIENTS OF LINEAR EQUATIONS REPRESENTING PATH SEGMENTS
C         AND REFERENCE LINES FOR EACH OBSTACLE

      IMPLICIT REAL*8 (A-H, O-Z)
      DIMENSION X(1000), Y(1000), BX(1000), BY(1000), A1(1000),
+      B1(1000), A2(1000), B2(1000), D2(1000)

      DO 4 I = 1, NOBS
        A1(I) = BY(I)
        B1(I) = -BX(I)
4      CONTINUE

      CX = 0.0
      CY = 0.0
      I = 1

      DO 5 J = 2, NUMPTS
        A2(I) = Y(J) - Y(I)
        B2(I) = X(I) - X(J)
        D2(I) = (Y(I)*X(J)) - (X(I)*Y(J))
        I = I + 1
5      CONTINUE

      RETURN
      END

```

```

      SUBROUTINE CASES1(A1, B1, A2, B2, D2, SEG, LINE, XINT, YINT)
C *****

```

```

C THIS SUBROUTINE DETERMINES THE COORDINATES FOR THE POINT OF
C INTERSECTION OF A GIVEN PATH SEGMENT AND A GIVEN OBSTACLE REFERENCE
C LINE. NUMERICAL STABILITY OF CALCULATIONS REQUIRES THE MATHEMATICAL
C OPERATIONS TO BE SEPARATED INTO CASES, THE BEST CASE BEING USED FOR
C EACH PARTICULAR SITUATION.

```

```

C INPUT: COEFFICIENTS OF LINEAR EQUATIONS FOR PATH SEGMENT AND
C REFERENCE LINE TO BE EVALUATED

```

```

C OUTPUT: POINT OF INTERSECTION OF THE PATH SEGMENT AND THE OBSTACLE
C REFERENCE LINE

```

```

      IMPLICIT REAL*8 (A-H,O-Z)
      INTEGER SEG
      DIMENSION A1(1000), B1(1000), A2(1000), B2(1000), D2(1000), A(2,2)

```

```

      A(1,1) = A1(LINE)
      A(1,2) = B1(LINE)
      A(2,1) = A2(SEG)
      A(2,2) = B2(SEG)
      BIGEST = 0.0

```

```

      DO 1 L = 1,2
        DO 1 K = 1,2
          TEST = DABS(A(K,L))
          IF (TEST.GT.BIGEST) THEN
            BIGEST = TEST
            KBIG = K
            LBIG = L
          ENDIF

```

```

1 CONTINUE

```

```

      IF (KBIG.EQ.1) THEN
        IF (LBIG.EQ.1) THEN
          YINT = -D2(SEG)/(B2(SEG)-B1(LINE)*A2(SEG)/A1(LINE))
          XINT = -B1(LINE)*YINT/A1(LINE)
          RETURN
        ELSE
          XINT = -D2(SEG)/(A2(SEG)-A1(LINE)*B2(SEG)/B1(LINE))
          YINT = -A1(LINE)*XINT/B1(LINE)
          RETURN
        ENDIF

```

```

      ELSE

```

```

        IF (LBIG.EQ.1) THEN
          YINT = (D2(SEG)*A1(LINE)/A2(SEG))/
+             (B1(LINE)-B2(SEG)*A1(LINE)/A2(SEG))
          XINT = (-D2(SEG)-B2(SEG)*YINT)/A2(SEG)
          RETURN

```

```

        ELSE
          XINT = (D2(SEG)*B1(LINE)/B2(SEG))/
+          (A1(LINE)-A2(SEG)*B1(LINE)/B2(SEG))
          YINT = (-D2(SEG)-XINT*A2(SEG))/B2(SEG)
          RETURN
        ENDIF
      ENDIF

    END

```

```

      SUBROUTINE ALPHAS (NEXT, LOLD, FSTSTR, LSTSTR, SEG, START, HEAD,
+      NSEGS, PRED)
C *****

```

```

C  GIVEN THAT A SEGMENT OF THE PATH CROSSES ONLY ALPHA RAYS WHEN
C  TRAVERSED, THESE CROSSINGS ARE SIMPLY INSERTED INTO THE LINK LIST
C  ARRAYS IN THE ORDER IN WHICH THEY WERE DETECTED, I.E., SMALLEST TO
C  LARGEST IN ABSOLUTE VALUE.

```

```

C  INPUT:  STRING REPRESENTING THE REFERENCE LINES CROSSED BY THE
C          CURRENT PATH SEGMENT, GIVEN THAT ALL CROSSINGS ARE ALPHAS

```

```

C  OUTPUT: UPDATED NEXT() AND PRED() ARRAYS WHICH CONTAIN THE STRING
C          REPRESENTING THE MOST CURRENT PATH SEGMENT

```

```

      INTEGER NEXT(1000), FSTSTR, SEG, START, HEAD(1000), PRED(1000)

      NEXT(LOLD) = FSTSTR
      PRED(FSTSTR) = LOLD
      LAST = LSTSTR - 1

      IF (LAST.GT.FSTSTR) THEN
        DO 12 I = FSTSTR, LAST
          NEXT(I) = I + 1
          PRED(I+1) = I
12      CONTINUE
      ELSE IF (LAST.EQ.FSTSTR) THEN
        NEXT(FSTSTR) = LSTSTR
        PRED(LSTSTR) = FSTSTR
      ENDIF

      IF (SEG.EQ.1) START = LOLD

      LOLD = LSTSTR
      RETURN
    END

```

```

      SUBROUTINE ORDER(SEG, LINE, HDINDX, HEAD, NEXT, A1, A2, B1, B2,
+          D2, FSTSTR, LSTSTR, X, Y, START, LENGTH, FIRST,
+          LOLD, NSEGS, PRED)
C *****

C  GIVEN THAT A SEGMENT OF THE PATH CROSSES ONE OR MORE BETAS, THIS
C  SUBROUTINE DETERMINES THE ACTUAL ORDER OF CROSSING WHEN TRAVERSING
C  THE SEGMENT FROM BEGINNING TO END.  THIS IS DONE BY FIRST DETERMINING
C  THE DISTANCE FROM THE SEGMENT START POINT TO THE POINT OF
C  INTERSECTION OF EACH CROSSED OBSTACLE REFERENCE LINE.  THESE
C  DISTANCES ARE THEN SORTED FROM SMALLEST TO LARGEST AND CROSSINGS
C  ARE UPDATED IN THE LINK LIST ACCORDINGLY.

C  INPUT:  HEAD() AND ALL COORDINATES REQUIRED TO DETERMINE THE
C          DISTANCES FROM INITIAL VERTEX OF PATH SEGMENT TO EACH OF
C          THE CROSSED REFERENCE LINES

C  OUTPUT: HEAD(), NEXT() AND PRED() ARRAYS CONTAINING THE RAW STRING
C          WHICH ACCURATELY REPRESENTS THE PATH ALONG THE CURRENT
C          SEGMENT

      INTEGER SEG, LINE, HEAD(1000), NEXT(1000), FSTSTR, HDINDX,
+          START, FIRST(1000), PRED(1000), F
      REAL*8 XINTER(1000), YINTER(1000), A1(1000), B1(1000),
+          A2(1000), B2(1000), D2(1000), DIST(1000), X(1000), Y(1000)

      DO 8 J = FSTSTR, LSTSTR
          LINE = ABS(HEAD(J))
          CALL CASES2 (A1, B1, A2, B2, D2, SEG, LINE, XINTER, YINTER)
          DIST(J) = (XINTER(LINE)-X(SEG))**2+(YINTER(LINE)-Y(SEG))**2
          DIST(J) = -DIST(J)
          NEXT(J) = J+1
          PRED(J+1) = J
      8 CONTINUE

      NEXT(LSTSTR) = 0
      PRED(LSTSTR+1) = LSTSTR

      CALL MERG2(DIST, NEXT, PRED, FSTSTR, F)

      NEXT(LOLD) = F
      PRED(F) = LOLD
      IF(SEG.EQ.1) START = LOLD
      LOLD = NEXT(LOLD)
17  IF(NEXT(LOLD).NE.0) THEN
          LOLD = NEXT(LOLD)
          GO TO 17
      ENDIF

      RETURN
      END

```

```

      SUBROUTINE CASES2(A1, B1, A2, B2, D2, SEG, LINE, XINTER, YINTER)
C *****
C DETERMINES COORDINATES FOR THE POINT OF INTERSECTION OF THE PATH
C SEGMENT AND EACH OF THE REFERENCE LINES IT CROSSES. AGAIN, IN ORDER
C TO MAINTAIN NUMERICAL STABILITY, CALCULATIONS ARE MADE USING THE
C MOST APPROPRIATE CLOSED FORM EQUATION.

C INPUT: COEFFICIENTS OF LINEAR EQUATIONS FOR CURRENT PATH SEGMENT
C        AND ALL OBSTACLE REFERENCE LINES WHICH IT CROSSES

C OUTPUT: COORDINATES FOR POINTS OF INTERSECTION OF EACH OBSTACLE
C         REFERENCE LINE WITH THE PATH SEGMENT

      IMPLICIT REAL*8 (A-H, O-Z)
      INTEGER SEG
      DIMENSION A1(1000), B1(1000), A2(1000), B2(1000), D2(1000),
+           XINTER(1000), YINTER(1000), A(2,2)

      A(1,1) = A1(LINE)
      A(1,2) = B1(LINE)
      A(2,1) = A2(SEG)
      A(2,2) = B2(SEG)
      BIGEST = 0.0

      DO 1 L = 1,2
        DO 1 K = 1,2
          TEST = DABS(A(K,L))
          IF (TEST.GT.BIGEST) THEN
            BIGEST = TEST
            KBIG = K
            LBIG = L
          ENDIF
        1 CONTINUE

      IF (KBIG.EQ.1) THEN
        IF (LBIG.EQ.1) THEN
          YINTER(LINE)= -D2(SEG)/(B2(SEG)-B1(LINE)*A2(SEG)/A1(LINE))
          XINTER(LINE)= -B1(LINE)*YINTER(LINE)/A1(LINE)
          RETURN
        ELSE
          XINTER(LINE)= -D2(SEG)/(A2(SEG)-A1(LINE)*B2(SEG)/B1(LINE))
          YINTER(LINE)= -A1(LINE)*XINTER(LINE)/B1(LINE)
          RETURN
        ENDIF
      ELSE
        IF (LBIG.EQ.1) THEN
          YINTER(LINE) = (D2(SEG)*A1(LINE)/A2(SEG))/
+           (B1(LINE)-B2(SEG)*A1(LINE)/A2(SEG))

```

```

        XINTER(LINE) = (-D2(SEG)-B2(SEG)*YINTER(LINE))/A2(SEG)
        RETURN
    ELSE
        XINTER(LINE) = (D2(SEG)*B1(LINE)/B2(SEG))/
+          (A1(LINE)-A2(SEG)*B1(LINE)/B2(SEG))
        YINTER(LINE) = (-D2(SEG)-XINTER(LINE)*A2(SEG))/B2(SEG)
        RETURN
    ENDIF
ENDIF
END

```

```

        SUBROUTINE MERG2 (DIST, NEXT, PRED, FSTSTR, F)
C *****

```

```

C THIS SUBROUTINE SORTS A SUBSTRING OF ALL POSTIVE INTEGERS INTO
C INCREASING ORDER.

C INPUT:  A DOUBLE LINK LIST CONSISTING OF DIST, NEXT, AND PRED ARRAYS

C OUTPUT: A DOUBLE LINK LIST WITH ALL ENTRIES PLACED IN THE ORDER
C         IN WHICH THEIR RESPECTIVE REFERENCE LINES WERE CROSSED

```

```

        IMPLICIT INTEGER(A-Z)
        REAL*8 DIST(1000)
        LOGICAL DONE
        DIMENSION NEXT(1000), PRED(1000)

        DONE = .FALSE.
        P = FSTSTR

1  F = P

        CALL SORT2 (F, PREDF, P, DIST, NEXT, DONE, PRED)

        IF(DONE) RETURN

        GOTO 1

        END

```

```

      SUBROUTINE SORT2(F, PREDF, P, DIST, NEXT, DONE, PRED)
C *****
C MARCHES DOWN A SUBSTRING OF POSITIVE INTEGERS AND ONLY SORTS IF
C ELEMENTS ARE NOT IN INCREASING ORDER. IF A NUMBER NEEDS TO BE
C PLACED HIGHER IN THE LIST SUBROUTINE 'PUT' IS CALLED TO DO SO
C INPUT: POINTER F INTO DOUBLE LINK LIST ARRAYS DIST, NEXT, AND PRED
C        TO INDICATE THE BEGINNING OF A SUBSTING OF POSITIVE INTEGERS,
C        AND PREDF
C OUTPUT: P IS A POINTER, DIST(P) IS THE LAST POSITIVE INTEGER IN THE
C         SUBSTRING THAT IS BEGUN BY DIST(F)

      IMPLICIT INTEGER (A-Z)
      REAL*8 DIST(1000)
      LOGICAL DONE
      DIMENSION NEXT(1000), PRED(1000)
      TAIL = F
1  NTAIL = NEXT(TAIL)

      IF(NTAIL.EQ.0) THEN
        DONE = .TRUE.
        RETURN
      ENDIF

      IF(DIST(NTAIL).LE.DIST(TAIL)) THEN
        TAIL = NTAIL
        GOTO 1
      ENDIF

      CALL PUT2(F, TAIL, NTAIL, PREDF, DIST, NEXT, PRED)

      GOTO 1

      END

```



```

      SUBROUTINE PUT2 (F, TAIL, NTAIL, PREDF, DIST, NEXT, PRED)
C *****
C REARRANGES POINTERS TO PLACE POSITIVE INTEGERS IN INCREASING ORDER.

C INPUT:  F      - START OF POSITIVE SUBSTRING
C         TAIL   - END OF CURRENTLY SORTED PORTION OF SUBSTRING
C         NTAIL = NEXT(TAIL) - POINTER TO THE SUCCESSOR OF TAIL IN THE
C                   ITEM TO BE INCORPORATED INTO THE SORTED PORTION OF
C                   THE LIST
C         PREDF  - THE PREDECESSOR OF F IN LINKED LIST

C OUTPUT: SOME POINTERS IN NEXT AND PRED ARE CHANGED TO PUT DIST(NTAIL)
C         IN ITS PROPER PLACE IN THE SORTED PORTION ON THE LIST

      IMPLICIT INTEGER (A-Z)
      REAL*8 DIST(1000)
      DIMENSION NEXT(1000), PRED(1000)

      IF (DIST(NTAIL).GE.DIST(F)) THEN
        TEMP = NEXT(NTAIL)
        NEXT(NTAIL) = F
        PRED(F) = NTAIL
        NEXT(TAIL) = TEMP
        PRED(TEMP) = TAIL
        F = NTAIL
      RETURN
    ENDIF
    CALL WALK2(F, TAIL, NTAIL, DIST, NEXT, PRED)
    RETURN
  END

```

```

      SUBROUTINE WALK2(F, TAIL, NTAIL, DIST, NEXT, PRED)
C *****
C THIS SUBROUTINE WALKS DOWN THE LINKED LIST AND PLACES THE DIST(NTAIL)
C IN THE CORRECT POSITION IN THE DIST ARRAY.
C INPUT: F, TAIL, NTAIL ARE USED AS ABOVE
C OUTPUT: ALTERS POINTERS IN NEXT AND PRED ARRAYS TO PLACE DIST(NTAIL)
C AFTER DIST(F) AND BEFORE DIST(TAIL) IN DOUBLE LINK LIST

      IMPLICIT INTEGER (A-Z)
      REAL*8 DIST(1000), HNTAIL
      DIMENSION NEXT(1000), PRED(1000)

      I = F
      NEXTI = NEXT(I)
      HNTAIL = DIST(NTAIL)

1 IF(HNTAIL.GE.DIST(NEXTI)) THEN
      NEXT(I) = NTAIL
      PRED(NTAIL) = I
      NNTAIL = NEXT(NTAIL)
      NEXT(NTAIL) = NEXTI
      PRED(NEXTI) = NTAIL
      NEXT(TAIL) = NNTAIL
      PRED(NNTAIL) = TAIL
      RETURN
ENDIF

      I = NEXTI
      NEXTI = NEXT(I)
      GOTO 1

      END

```

```

      SUBROUTINE COUNTR(START, NEXT, HEAD, NELEMS)
C *****
C THIS SUBROUTINE COUNTS THE NUMBER OF ELEMENTS IN A GIVEN STRING OF
C CHARACTERS
C INPUT:  HEAD() AND NEXT() ARRAYS FOR THE STRING OF CHARACTERS
C OUTPUT: NUMBER OF ELEMENTS IN THE STRING
      INTEGER PTR, NEXT(1000), HEAD(1000), START
      PTR = NEXT(START)
      NELEMS = 0
20  IF (PTR.NE.0) THEN
      NELEMS = NELEMS + 1
      PTR = NEXT(PTR)
      GO TO 20
    ENDIF
      RETURN
      END

```

```

      SUBROUTINE ALG2(NOBS, NELEMS, HEAD, NEXT, X, Y, BX, BY, FR, NR,
+          PR)
C *****
C  GENERATES THE WELL-KNOW FUNDAMENTAL GROUP REPRESENTATION OF AN
C  EQUIVALENCE CLASS.

C  INPUT:  RAW OR CANONICAL STRING, WITH # OBSTACLES, # ELEMENTS IN
C          STRING, COORDINATES OF A AND OBSTACLES (BK)

C  OUTPUT: 1) IF INPUT IS RAW STRING, THEN POSSIBLY UNREDUCED
C           FUNDAMENTAL GROUP WILL RESULT (IF CANCELLATION OF LIKE
C           POSITIVE NUMBERS COULD HAVE OCCURRED IN THE RAW STRING).
C           2) IF INPUT IS CANONICAL STRING, THEN THE RESULTING
C           FUNDAMENTAL GROUP WILL BE IN REDUCED FORM.

      REAL*8  XA, YA, XB, YB, X(1000), Y(1000), BX(1000), BY(1000)
      INTEGER NOBS, NELEMS, HEAD(1000), NEXT(1000), PRED(1000),
+          FR(1000), NR(1000), PR(1000), S(1000), FUNDGP(1000),
+          START
      LOGICAL RIGHT(1000), RITE

      LENGTH = 1
      START = NEXT(1)

      XA = X(1)
      YA = Y(1)

      DO 1 K = 1, NOBS
        XB = BX(K)
        YB = BY(K)
1      RIGHT(K) = RITE(XA, YA, XB, YB)

      NELEM = NELEMS + 1
      DO 10 M = 2, NELEM
        S(M) = HEAD(START)
        J = IABS(S(M))
        RIGHT(J) = .NOT. RIGHT(J)
        IF(S(M).GT.0) THEN
          LENGTH = LENGTH + 1
          IF(RIGHT(J)) THEN
            FUNDGP(LENGTH) = J
          ELSE
            FUNDGP(LENGTH) = -J
          ENDIF
        ENDIF
        START = NEXT(START)
        IF(START.EQ.0) GOTO 11
10  CONTINUE

```

```

11 FR(1) = 0
   NR(1) = 2

   DO 2 I = 2, LENGTH
       FR(I) = FUNDGP(I)
       NR(I) = I + 1
       PR(I) = I - 1
2 CONTINUE

NR(LENGTH) = 0

RETURN
END

```

```

      LOGICAL FUNCTION RITE(XA, YA, XB, YB)
C *****
C THIS FUNCTION DETERMINES WHICH SIDE OF A GIVEN DIRECTED LINE ANY
C POINT LIES.
C INPUT: TWO POINTS THAT DETERMINE THE LINE
C OUTPUT: LOGICAL VARIABLE THAT IS TRUE IF A POINT LIES TO THE RIGHT
C         AND FALSE IF A POINT LIES TO THE LEFT

      REAL*8  XB, YB, XA, YA, SIGNA

      RITE = .TRUE.

      SIGNA = -((YB*XA)-(XB*YA))

      IF(SIGNA.LT.0)THEN
          RITE = .FALSE.
          RETURN
      ENDIF

      IF(SIGNA.EQ.0) THEN
          PRINT*, 'THE POINT A LIES ON THE LINE LK PROGRAM STOPS'
          STOP
      ENDIF

      RETURN
      END

```

```

      SUBROUTINE CANALG2(HEAD, NEXT, PRED)
C *****
C THIS SUBROUTINE TAKES AN UNREDUCED FUNDAMENTAL GROUP REPRESENTATION
C OF A GIVEN CLASS AND CANCELS A GENERATOR IF IT IS ADJACENT TO ITS
C INVERSE.

C INPUT:  HEAD, NEXT, AND PRED ARRAYS

C OUTPUT: HEAD, NEXT, AND PRED ARRAYS WITH NEXT AND PRED REARRANGED
C         TO SKIP AROUND CANCELLED ELEMENTS

      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      START = 1
      PTR1 = START
      PTR2 = START
      PTR3 = NEXT(PTR2)
10  IF (PTR3.NE.0) THEN
      IF (HEAD(PTR2).EQ.-(HEAD(PTR3))) THEN
        NEXT(PTR1) = NEXT(PTR3)
        PRED(NEXT(PTR3)) = PTR1
        IF(NEXT(PTR1).EQ.0) RETURN
        PTR2 = PTR1
        PTR1 = PRED(PTR1)
        PTR3 = NEXT(PTR2)
      ELSE
        PTR1 = PTR2
        PTR2 = PTR3
        PTR3 = NEXT(PTR2)
      ENDIF
      GO TO 10
    ENDIF
  RETURN
  END

```

```

      SUBROUTINE ALG1(HEAD, NEXT, PRED, NELEMS)
C *****
C THIS SUBROUTINE TAKES A GIVEN RAW STRING OF CHARACTERS REPRESENTING
C A PATH AND PRODUCES THE CANONICAL FORM OF THAT STRING. THIS IS DONE
C BY FIRST ORDERING ALL OF THE ALPHA SUBSTRINGS FROM SMALLEST TO
C LARGEST IN ABSOLUTE VALUE. NEXT, CANCELLATION IS PERFORMED TO
C ELIMINATE ALL LIKE PAIRS OF ADJACENT ELEMENTS FROM THE STRING.
C INPUT:  RAW STRING IN FORM OF DOUBLE LINKED LIST WITH HEAD(), NEXT()
C         AND PRED() ARRAYS
C OUTPUT: CANONICAL FORM OF THE RAW STRING
      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)
      START = 1
      CALL MERG1(HEAD, NEXT, PRED)
      CALL CANCEL(START, HEAD, NEXT, PRED, NELEMS)
      RETURN
      END

```

```

      SUBROUTINE MERG1 (HEAD, NEXT, PRED)
C *****
C THIS SUBROUTINE DOES THE INITIAL SORT OF THE ALPHA SUBSTRINGS IN THE
C RAWSTRING (INCREASING IN ABSOLUTE VALUE).
C INPUT:  HEAD, NEXT, PRED ARRAYS REPRESENTING A DOUBLE LINKED LIST
C         OF THE RAWSTRING WITH THE ALPHA SUBSTRINGS UNORDERED
C OUTPUT: POINTERS STORED IN THE ARRAYS NEXT AND PRED ARE ALTERED SO
C         THAT EACH SUBSTRING OF THE STORED LIST WHICH CONSISTS
C         ENTIRELY OF NEGATIVE INTEGERS IS SORTED INTO NON-INCREASING
C         ORDER, WHILE SUBSTRINGS OF POSITIVE INTEGERS ARE LEFT
C         UNALTERED.

      IMPLICIT INTEGER(A-Z)
      LOGICAL DONE
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      DONE = .FALSE.
      P = 1

1 CALL FRONT (P, F, PREDF, HEAD, NEXT, DONE)

      IF(DONE) RETURN

      CALL SORT (F, PREDF, P, HEAD, NEXT, DONE, PRED)

      IF(DONE) RETURN

      GOTO 1

      END

```



```

      SUBROUTINE FRONT(P, F, PREDF, HEAD, NEXT, DONE)
C *****
C  FINDS THE BEGINNING OF NEGATIVE INTEGER STRINGS (ALPHA STRING)
C  INPUT:  POINTER P INTO LINKED LIST
C  OUTPUT: F IS A POINTER INTO THE HEAD ARRAY.  POINTS TO FIRST NEGATIVE
C          ENTRY WHICH OCCURS STRICTLY AFTER HEAD(P).  PREDF IS POINTER
C          SUCH THAT HEAD(PREDF) IS THE PREDESSOR OF HEAD(F).

      IMPLICIT INTEGER(A-Z)
      LOGICAL DONE
      DIMENSION HEAD(1000), NEXT(1000)

      F = P
1  PREDF = F
      F = NEXT(F)

      IF (F.EQ.0) THEN
         DONE = .TRUE.
         RETURN
      ENDIF

      IF (HEAD(F).LT.0) RETURN

      GOTO 1

      END

```

```

      SUBROUTINE SORT(F, PREDF, P, HEAD, NEXT, DONE, PRED)
C *****
C  MARCHES DOWN A SUBSTRING OF NEGATIVE INTEGERS AND ONLY SORTS IF
C  ELEMENTS ARE IN INCREASING ORDER.  IF A NUMBER NEEDS TO BE
C  PLACED HIGHER IN THE LIST SUBROUTINE 'PUT' IS CALLED TO DO SO
C
C  INPUT:  POINTER F INTO DOUBLE LINK LIST ARRAYS HEAD, NEXT, AND PRED
C          TO INDICATE THE BEGINNING OF A SUBSTRING OF NEGATIVE
C          INTEGERS; AND PREDF
C
C  OUTPUT: P IS A POINTER, HEAD(P) IS THE LAST NEGATIVE INTEGER IN THE
C          SUBSTRING THAT IS BEGUN BY HEAD(F)
C
      IMPLICIT INTEGER (A-Z)
      LOGICAL DONE
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)
C
      TAIL = F
1  NTAIL = NEXT(TAIL)
C
      IF(NTAIL.EQ.0) THEN
        DONE = .TRUE.
        RETURN
      ENDIF
C
      IF(HEAD(NTAIL).GT.0) THEN
        P = TAIL
        RETURN
      ENDIF
C
      IF(HEAD(TAIL).LE.HEAD(NTAIL)) THEN
        TAIL = NTAIL
        GOTO 1
      ENDIF
C
      CALL PUT(F, TAIL, NTAIL, PREDF, HEAD, NEXT, PRED)
C
      GOTO 1
C
      END

```

```

      SUBROUTINE PUT (F, TAIL, NTAIL, PREDF, HEAD, NEXT, PRED)
C *****

C SUBROUTINE THAT REARRANGES POINTERS TO PLACE A NEGATIVE INTEGER IN
C NON-INCREASING ORDER.

C INPUT:  F      - START OF NEGATIVE SUBSTRING
C          TAIL   - END OF CURRENTLY SORTED PORTION OF SUBSTRING
C          NTAIL  = NEXT(TAIL) - POINTER TO THE SUCCESSOR OF TAIL IN THE
C                   ITEM TO BE INCORPORATED INTO THE SORTED PORTION OF
C                   THE LIST.
C          PREDF  - THE PREDECESSOR OF F IN LINKED LIST

C OUTPUT: SOME POINTERS IN NEXT AND PRED ARE CHANGED TO PUT HEAD(NTAIL)
C          IN ITS PROPER PLACE IN THE SORTED PORTION ON THE LIST

      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      IF (HEAD(NTAIL).GE.HEAD(F)) THEN
        NEXT(PREDF) = NTAIL
        PRED(NTAIL) = PREDF
        TEMP = NEXT(NTAIL)
        NEXT(NTAIL) = F
        PRED(F) = NTAIL
        NEXT(TAIL) = TEMP
        PRED(TEMP) = TAIL
        F = NTAIL
        RETURN
      ENDIF

      CALL WALK(F, TAIL, NTAIL, HEAD, NEXT, PRED)

      RETURN
      END

```

```

      SUBROUTINE WALK(F, TAIL, NTAIL, HEAD, NEXT, PRED)
C *****
C THIS SUBROUTINE WALKS DOWN THE LINKED LIST AND PLACES THE HEAD(NTAIL)
C IN THE CORRECT POSITION IN THE HEAD ARRAY. (DECREASING ORDER)
C INPUT: F, TAIL, NTAIL ARE USED AS ABOVE
C OUTPUT: ALTERS POINTERS IN NEXT AND PRED ARRAYS TO PLACE HEAD(NTAIL)
C          AFTER HEAD(F) AND BEFORE HEAD(TAIL) IN DOUBLE LINK LIST

      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      I = F
      NEXTI = NEXT(I)
      HNTAIL = HEAD(NTAIL)

1  IF(HNTAIL.GE.HEAD(NEXTI)) THEN
      NEXT(I) = NTAIL
      PRED(NTAIL) = I
      NNTAIL = NEXT(NTAIL)
      NEXT(NTAIL) = NEXTI
      PRED(NEXTI) = NTAIL
      NEXT(TAIL) = NNTAIL
      PRED(NNTAIL) = TAIL
      RETURN
  ENDIF

      I = NEXTI
      NEXTI = NEXT(I)
      GOTO 1

  END

```

```

      SUBROUTINE CANCEL(START, HEAD, NEXT, PRED, NELEMS)
C *****

C  GIVEN THE SORTED RAW STRING, THIS SUBROUTINE REDUCES THE STRING TO
C  CANONICAL FORM BY ADJUSTING THE LINK LIST TO SKIP ANY PAIRS OF
C  ADJACENT LIKE ELEMENTS IN THE STRING.  ONCE TWO ELEMENTS ARE
C  ELIMINATED FROM THE STRING, THE SUBSTRINGS WHICH WERE ON THEIR LEFT
C  AND RIGHT CONCATENATE TO FORM A NEW STRING AND HENCE A NEW PAIR OF
C  ADJACENT ELEMENTS WHICH MUST ALSO BE CHECKED FOR EQUALITY.  IN THE
C  EVENT A BETA SUBSTRING BECOMES ANNIHILATED FROM THE STRING, THE TWO
C  ADJACENT ALPHA STRINGS CONCATENATE AND ARE AGAIN SORTED FROM SMALLEST
C  TO LARGEST IN ABSOLUTE VALUE AS A SINGLE SUBSTRING.

C  INPUT:  SORTED RAW STRING

C  OUTPUT: CANONICAL FORM OF THE RAW STRING

      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      START = 1
      START1 = START
      START2 = START
      BETA = START
      PTR1 = START1
      PTR2 = START1
50  PTR3 = NEXT(PTR2)

      IF (HEAD(PTR3).LT.0) THEN
          START1 = PTR3
          START2 = START1
      ELSE
          BETA = PTR3
      ENDIF

10  IF (PTR3.NE.0) THEN
      CALL CHEKER (HEAD, START1, START2, PTR1, PTR2, PTR3, BETA)
      IF (HEAD(PTR2).EQ.HEAD(PTR3)) THEN
          NEXT(PTR1) = NEXT(PTR3)
          PRED(NEXT(PTR3)) = PTR1
          PTR2 = NEXT(PTR3)
          IF (PTR2.EQ.0) GO TO 60
          CHECK1 = HEAD(PTR2) * HEAD(PTR3)
          CHECK2 = HEAD(PTR1) * HEAD(PTR2)
          IF (CHECK1.LT.0.AND.CHECK2.GT.0.AND.HEAD(PTR2).LT.0)THEN
              PTR2 = PTR1
              PTR1 = PRED(PTR1)
              PTR3 = NEXT(PTR2)
              START1 = START2
6          IF (START1.NE.1.AND.HEAD(PRED(START1)).LT.0)THEN

```

```

        START1 = PRED(START1)
        GO TO 6
    ENDIF
    START2 = PTR3
    CALL MERGE (HEAD, NEXT, START1, START2, PRED)
    PTR1 = PRED(START1)
    PTR2 = PTR1
    GO TO 50
+   ELSEIF (CHECK1.LT.0.AND.CHECK2.EQ.0.AND.HEAD(PTR2).LT.0)
    THEN
        PTR2 = PTR1
        PTR3 = NEXT(PTR1)
        START1 = PTR3
        START2 = START1
        GO TO 10
+   ELSEIF (CHECK1.GT.0.AND.CHECK2.GT.0.AND.HEAD(PTR2).GT.0)
    THEN
        IF (BETA.EQ.2) THEN
            BETA = 1
        ELSEIF (BETA.EQ.1) THEN
            GO TO 9
        ELSE
11          IF(START1.NE.1) BETA = PRED(START1)
            IF(HEAD(PRED(BETA)).GT.0) THEN
                BETA = PRED(BETA)
                GO TO 11
            ENDIF
        ENDIF
9        PTR1 = PRED(BETA)
        PTR2 = BETA
        PTR3 = NEXT(PTR2)
        GO TO 10
+   ELSEIF (CHECK1.LT.0.AND.CHECK2.GT.0.AND.HEAD(PTR2).GT.0)
    THEN
        PTR2 = PTR1
        PTR1 = PRED(PTR1)

        IF(HEAD(PTR1).LT.0)THEN
            START1 = PTR1
5            IF (HEAD(PRED(START1)).LT.0)THEN
                START1 = PRED(START1)
                GO TO 5
            ENDIF
        ENDIF
        PTR3 = NEXT(PTR2)
        GO TO 10
+   ELSEIF (CHECK1.LT.0.AND.CHECK2.LT.0.AND.HEAD(PTR2).LT.0)
    THEN
        IF(START1.EQ.1) THEN
            START1 = PTR2

```

```

        START2 = PTR2
        PTR3 = NEXT(PTR2)
    ELSE
        START1 = START2
        PTR3 = NEXT(PTR2)
    ENDIF
    GO TO 10
ELSE
    START1 = START2
    PTR3 = NEXT(PTR2)
ENDIF
ELSE
    PTR1 = PTR2
    PTR2 = PTR3
    PTR3 = NEXT(PTR2)
ENDIF
GO TO 10
ENDIF

60 CALL COUNTR (START, NEXT, HEAD, NELEMS)

RETURN
END

```

```

      SUBROUTINE CHEKER(HEAD, START1, START2, PTR1, PTR2, PTR3, BETA)
C *****

C  AS POINTERS MOVE ALONG THE CHARACTER STRING DURING CANCELLATION THIS
C  SUBROUTINE CHECKS TO DETERMINE WHETHER OR NOT THE END OF ONE
C  SUBSTRING IS REACHED AND A NEW ONE BEGINS.  DEPENDING ON THE OUTCOME
C  OF THIS CHECK, THE POINTERS USED TO IDENTIFY THE BEGINNING OF THE
C  TWO LATEST ALPHA STRINGS AND THE LATEST BETA STRING ARE UPDATED

C  INPUT:  HEAD() ARRAY AND PRESENT POINTER LOCATIONS FROM 'CANCEL'
C          SUBROUTINE

C  OUTPUT: MODIFIED INDICES FOR LOCATION OF ALPHA AND BETA STRINGS

      INTEGER HEAD(1000), START1, START2, PTR1, PTR2, PTR3, CHECK, BETA

      CHECK = HEAD(PTR3) * HEAD(PTR2)

      IF (CHECK.GT.0) THEN
        IF (HEAD(PTR2).LT.0.AND.HEAD(PTR1).GT.0) START2 = PTR2
        IF (HEAD(PTR2).GT.0.AND.HEAD(PTR1).LT.0) BETA = PTR2
      ELSEIF (CHECK.LE.0) THEN
        IF (HEAD(PTR2).GT.0.AND.HEAD(PTR1).LT.0) THEN
          BETA = PTR2
          START1 = START2
          START2 = PTR3
        ELSEIF (HEAD(PTR2).GT.0) THEN
          IF (START1.NE.1) THEN
            START1 = START2
            START2 = PTR3
          ELSE
            START1 = PTR3
            START2 = START1
          ENDIF
        ELSEIF (HEAD(PTR2).LT.0.AND.HEAD(PTR1).GT.0) THEN
          BETA = PTR3
          START1 = START2
          START2 = PTR2
        ELSEIF (HEAD(PTR2).LT.0.AND.HEAD(PTR1).LT.0) THEN
          BETA = PTR3
        ENDIF
      ENDIF

      RETURN
      END

```



```

SUBROUTINE MERGE(HEAD, NEXT, START1, START2, PRED)
C *****

C THIS SUBROUTINE MERGES TWO ORDERED ALPHA STRINGS. THE RESULTING
C SUBSTRING WILL BE INCREASING IN ABSOLUTE VALUE.

C INPUT: HEAD(), NEXT() AND PRED() ARRAYS ALONG WITH INDICES START1
C AND START2 INDICATING BEGINNING OF THE ALPHA STRINGS TO BE
C MERGED

C OUTPUT: MODIFIED STRING CONTAINING THE MERGED ALPHA STRINGS WHICH
C WERE A RESULT OF ANNIHILATION OF A BETA STRING

      INTEGER HEAD(1000), NEXT(1000), START1, START2, P1, P2, TAIL,
+         PRED(1000)
      LOGICAL GO

      IF (START1.EQ.1) GO TO 2

      IF(HEAD(PRED(START1)).EQ.0) START1 = NEXT(PRED(START1))

      IF(NEXT(PRED(START1)).NE.START1)THEN
        START1 = NEXT(PRED(START1))
7      IF (START1.EQ.1) GO TO 2
        IF(HEAD(START1).GT.0)THEN
          START1 = NEXT(START1)
          GO TO 7
        ENDIF
8      IF(HEAD(PRED(START1)).LT.0)THEN
          START1 = PRED(START1)
          GO TO 8
        ENDIF
      ENDIF

2 CALL SET(HEAD, NEXT, START1, START2, P1, P2, TAIL, PRED)

1 IF(GO(P1, P2, START2, HEAD, NEXT)) THEN
      CALL MERGER(HEAD, NEXT, P1, P2, TAIL, PRED)
      GOTO 1
    ENDIF

      CALL FXTAIL(HEAD, NEXT, P1, P2, TAIL, START2, PRED)

      RETURN
      END

```

```

      SUBROUTINE SET(HEAD, NEXT, START1, START2, P1, P2, TAIL, PRED)
C *****
C  SETS THE STARTING POINTER OF THE NEW STRING AND THE TWO POINTERS
C  OF THE STRINGS TO BE MERGED
C  INPUT:  POINTERS INTO TWO ALPHA STRINGS
C  OUTPUT: POINTER INTO BEGINNING OF A NEW SORTED STRING

      INTEGER HEAD(1000), NEXT(1000), START1, START2, P1, P2, TAIL,
+         PRED(1000)

      P1 = START1
      P2 = START2
      IF(HEAD(P1).GT.HEAD(P2)) THEN
        TAIL = P1
        START1 = P1
        P1 = NEXT(P1)
      ELSE
        TAIL = P2
        START1 = P2
        NEXT(PRED(P1)) = P2
        PRED(P2) = PRED(P1)
        P2 = NEXT(P2)
      ENDIF

      RETURN
      END

```

```

      LOGICAL FUNCTION GO(P1, P2, START2, HEAD, NEXT)
C *****
C LOGICAL FUNCTION TO DETERMINE WHEN THE END OF EITHER LIST IS REACHED
C INPUT:  POINTERS INTO TWO ALPHA SUBSTRINGS
C OUTPUT: LOGICAL VARIABLE WHICH IS TRUE IF EITHER POINTER IS AT THE
C         END OF A STRING, AND FALSE OTHERWISE

      INTEGER HEAD(1000), NEXT(1000), P1, P2, START2

      GO = .TRUE.

      IF(P1.EQ.START2)THEN
        GO = .FALSE.
        RETURN
      ENDIF

      IF(NEXT(P2).EQ.0)THEN
        GO = .FALSE.
        RETURN
      ENDIF

      IF(HEAD(P2).GT.0)THEN
        GO = .FALSE.
        RETURN
      ENDIF

      RETURN
      END

```

```

      SUBROUTINE MERGER(HEAD, NEXT, P1, P2, TAIL, PRED)
C *****

C THIS SUBROUTINE COMPARES TWO ELEMENTS - ONE IN EACH ALPHA SUBSTRING
C AND PLACES THE ONE THAT IS THE SMALLEST ON THE LIST REPRESENTING THE
C NEW SORTED LIST

C INPUT: HEAD, NEXT, PRED, AND POINTERS INTO THE HEAD ARRAY

C OUTPUT: HEAD, NEXT, PRED, AND POINTERS FOR THE NEXT TWO ELEMENTS
C         TO BE CHECKED

      INTEGER HEAD(1000), NEXT(1000), P1, P2, TAIL , PRED(1000)

      IF(HEAD(P1).GT.HEAD(P2)) THEN
        CALL TACON1(NEXT, P1, TAIL, PRED)
      ELSE
        CALL TACON2(NEXT, P2, TAIL, PRED)
      ENDIF

      RETURN
      END

```

```

      SUBROUTINE TACON1 (NEXT, P1, TAIL, PRED)
C *****

C ADDS THE ELEMENT TO WHICH P1 POINTS TO THE TAIL OF THE NEW STRING

C INPUT: NEXT, PRED, P1, TAIL

C OUTPUT: NEXT, PRED, P1, TAIL

      INTEGER NEXT(1000), P1, TAIL, PRED(1000)

      NEXT(TAIL) = P1
      PRED(P1) = TAIL
      TAIL = P1
      P1 = NEXT(P1)

      RETURN
      END

```

```

      SUBROUTINE TACON2 (NEXT, P2, TAIL, PRED)
C *****
C  ADDS THE ELEMENT TO WHICH P2 POINTS TO THE TAIL OF THE NEW STRING
C  INPUT:  NEXT, PRED, P2, TAIL
C  OUTPUT: NEXT, PRED, P2, TAIL

      INTEGER NEXT(1000), P2, TAIL, PRED(1000)

      NEXT(TAIL) = P2
      PRED(P2) = TAIL
      TAIL = P2
      P2 = NEXT(P2)
      RETURN
      END

```

```

      SUBROUTINE FXTAIL(HEAD, NEXT, P1, P2, TAIL, START2, PRED)
C *****
C ATTACHES THE END OF THE LONGER SORTED STRING TO THE END OF THE MERGED
C STRING
C INPUT: P1, P2, HEAD, NEXT, PRED, TAIL
C OUTPUT: P1, P2, HEAD, NEXT, PRED, TAIL, START2
      INTEGER HEAD(1000), NEXT(1000), P1, P2, TAIL, START2, P11,
+       PRED(1000)
      IF(P1.EQ.START2)THEN
        CALL TACON2(NEXT, P2, TAIL, PRED)
        RETURN
      ENDIF
      IF(NEXT(P2).EQ.0)THEN
        CALL TACON1(NEXT, P1, TAIL, PRED)
        P11 = TAIL
2       IF(P1.NE.START2)THEN
          P11 = P1
          P1 = NEXT(P1)
          GOTO 2
        ENDIF
        NEXT(P11) = P2
        PRED(P2) = P11
        RETURN
      ENDIF
      IF(HEAD(P2).GT.0)THEN
        CALL TACON1(NEXT, P1, TAIL, PRED)
        P11 = TAIL
3       IF(P1.NE.START2) THEN
          P11 = P1
          P1 = NEXT(P1)
          GOTO 3
        ENDIF
        NEXT(P11) = P2
        PRED(P2) = P11
        RETURN
      ENDIF
      RETURN
      END

```

```

      SUBROUTINE CTREX(FL, NL, PL, FR, NR, PR, BSEED, PSEED)
C *****
C  THIS SUBROUTINE CHECKS F(C(R(P))) AGAINST K(F(R(P))) FOR EQUALITY.
C  INPUT:  FUNDAMENTAL GROUP REPRESENTATIONS FOR BOTH ALGORITHMS
C  OUTPUT: MESSAGE INDICATING THE OCCURRENCE OF UNEQUAL FUNDAMENTAL
C           GROUPS

      INTEGER FL(1000), NL(1000), PL(1000), FR(1000), NR(1000), PR(1000)

      IF(NL(1).EQ.0.AND.NR(1).EQ.0) GO TO 11

      I = NL(1)
      J = NR(1)

10  IF (I.EQ.0.AND.J.EQ.0) GO TO 11

      IF (FL(I).EQ.0.AND.FR(J).EQ.0) GO TO 11

      IF (FL(I).EQ.FR(J)) THEN
        I = NL(I)
        J = NR(J)
        GOTO 10
      ELSE
        PRINT*, 'STRINGS UNEQUAL'
        PRINT*, ' '
        IF(NL(1).EQ.0)THEN
          PRINT*, 'THE GROUP IS EMPTY AFTER ALG1 SEQUENCE'
        ELSE
          PRINT*, 'THE FUNDAMENTAL GROUP AFTER ALG1 IS:'
          CALL PRINTS(FL,NL)
        ENDIF
        IF(NR(1).EQ.0) THEN
          PRINT*, 'THE GROUP IS EMPTY AFTER ALG2 SEQUENCE'
        ELSE
          PRINT*, 'THE FUNDAMENTAL GROUP AFTER ALG2 IS:'
          CALL PRINTS(FR,NR)
        ENDIF
        PRINT*, ' '
        PRINT*, 'BSEED AND PSEED ARE',BSEED,PSEED

      ENDIF

11  RETURN
      END

```

```

      SUBROUTINE PRINTS(HEAD, NEXT)
C *****
C THIS SUBROUTINE CAN BE USED TO PRINT OUT ANY STRING THAT IS STORED IN
C LINK LIST FORM
C INPUT:  LINK LIST ARRAYS
C OUTPUT: HORIZONTAL STRING OF THE ELEMENTS ACCORDING TO THE NEXT()

      INTEGER HEAD(1000), NEXT(1000), STRING(1000)

      NSTART = NEXT(1)
      I = 0

      98 IF(NSTART.NE.0) THEN
          I = I + 1
          STRING(I) = HEAD(NSTART)
          NSTART = NEXT(NSTART)
          GOTO 98
      ENDIF

      PRINT 111, (STRING(J), J = 1, I)
111 FORMAT(' ',20I4)
      PRINT*, ' '

      RETURN
      END

```



```

C ALL SUBROUTINES BELOW THIS POINT WERE USED AS DEBUG TOOLS. THEY
C GENERATE CRUDE GRAPHS WHICH PLOT THE BOARDS AND THE PATHS.
C THE VALUES ACQUIRED BY THE RAWSTRING , MERGE, AND MANY OTHER
C SUBROUTINES WERE CHECKED USING THESE ROUTINES.
C *****

```

```

      SUBROUTINE GRAPH(BX, BY, X, Y, NOBS, NUMPTS)
C *****

```

```

      REAL*8 X(1000), Y(1000), BX(1000), BY(1000)
      CHARACTER*1 MATR(53, 105)

```

```

      CALL GRAFPA(X, Y, NUMPTS, MATR)
      CALL GRAFOB(BX, BY, NOBS, MATR)

```

```

      DO 1 I = 1,53
1      PRINT 111, (MATR(I,J), J=1,105)
111  FORMAT(' ',105A1)

```

```

      RETURN
      END

```

```

      SUBROUTINE GRAFPA (X, Y, NUMPTS, MATR)
C *****
      REAL*8 X(1000), Y(1000)
      CHARACTER*1 MATR(53, 105), CH

      DO 10 L = 2,52
        DO 10 M = 2,104
10          MATR(L,M) = ' '

      DO 1 I = 1, 105
        MATR(1,I) = '*'
1          MATR(53,I) = '*'

      DO 4 I = 1, 53
        MATR(I,1) = '*'
4          MATR(I,105) = '*'

      XK = X(1)
      YK = Y(1)
      CALL COOR(XK, YK, IX, IY)
      CALL CHARPA(1, CH)
      MATR(IY,IX) = CH

      DO 2 K = 2,NUMPTS
2        CALL FILL(K, X, Y, MATR)

      DO 3 K = 2,NUMPTS
        XK = X(K)
        YK = Y(K)
        CALL COOR(XK, YK, IX, IY)
        CALL CHARPA(K, CH)
3        MATR(IY,IX) = CH

      RETURN
      END

```

```

      SUBROUTINE COOR(BX, BY, IX, IY)
C *****
      REAL*8 BX, BY

      IX = MESHP(BX) * 2
      IY = 54 - MESHP(BY)

      RETURN
      END

```

```

      INTEGER FUNCTION MESHP(X)
C *****
      REAL*8 X

      MESHP = ((X+1.)*(51./2.)) + 2.

      RETURN
      END

```

```

      SUBROUTINE CHARPA(K, CH)
C *****
      CHARACTER*1 CH

      IF(K.EQ.1) THEN
        CH = 'A'
        RETURN
      END IF

      IF(K.EQ.2) THEN
        CH = 'B'
        RETURN
      END IF

      IF(K.EQ.3) THEN
        CH = 'C'
        RETURN
      END IF

      IF(K.EQ.4) THEN
        CH = 'D'
        RETURN
      END IF

      IF(K.EQ.5) THEN
        CH = 'E'
        RETURN
      END IF

      IF(K.EQ.6) THEN
        CH = 'A'
        RETURN
      END IF

      END

```

```

      SUBROUTINE FILL(K, X, Y, MATR)
C *****
      REAL*8 X(1000), Y(1000), U, V, DX, DY, XK, YK, XKM1, YKM1
      CHARACTER*1 MATR(53,105)

      XK = X(K)
      YK = Y(K)
      XKM1 = X(K-1)
      YKM1 = Y(K-1)

      CALL COOR(XK, YK, KX, KY)

      CALL COOR(XKM1, YKM1, KXM1, KYM1)

      L = IABS(KX - KXM1) - 1
      M = IABS(KY - KYM1) - 1
      IF (M.GT.L) L = M
      DX = (XK - XKM1)/(L+1)
      DY = (YK - YKM1)/(L+1)
      DO 1 J = 1,L
          U = XKM1 + J*DX
          V = YKM1 + J*DY
          CALL COOR(U,V, IU, IV)
1      MATR(IV, IU) = '+'

      RETURN
      END

```

```

      SUBROUTINE GRAFOB (X, Y, NOBS, MATR)
C *****
      REAL*8 X(1000), Y(1000)
      CHARACTER*1 MATR(53, 105), CH

      MATR(27, 53) = '*'

      DO 2 K = 1,NOBS
          XK = X(K)
          YK = Y(K)
          CALL COOR(XK, YK, IX, IY)
          CALL CHAR(K, CH)
2      MATR(IY, IX) = CH

      RETURN
      END

```

```

SUBROUTINE CHAR(K, CH)
C *****
CHARACTER*1 CH

IF(K.EQ.1) THEN
    CH = '1'
    RETURN
END IF

IF(K.EQ.2) THEN
    CH = '2'
    RETURN
END IF

IF(K.EQ.3) THEN
    CH = '3'
    RETURN
ENDIF

IF(K.EQ.4) THEN
    CH = '4'
    RETURN
ENDIF

IF(K.EQ.5) THEN
    CH = '5'
    RETURN
ENDIF

IF(K.EQ.6) THEN
    CH = '6'
    RETURN
END IF

IF(K.EQ.7) THEN
    CH = '7'
    RETURN
END IF

IF(K.EQ.8) THEN
    CH = '8'
    RETURN
ENDIF

IF(K.EQ.9) THEN
    CH = '9'
    RETURN
ENDIF

IF(K.EQ.10) THEN
    CH = '@'
    RETURN

```

```
ENDIF

IF(K.EQ.11) THEN
    CH = '#'
    RETURN
END IF

IF(K.EQ.12) THEN
    CH = '$'
    RETURN
END IF

IF(K.EQ.13) THEN
    CH = '%'
    RETURN
ENDIF

IF(K.EQ.14) THEN
    CH = '&'
    RETURN
ENDIF

IF(K.EQ.15) THEN
    CH = 'φ'
    RETURN
ENDIF

IF(K.EQ.16) THEN
    CH = '/'
    RETURN
END IF

IF(K.EQ.17) THEN
    CH = '\'
    RETURN
END IF

IF(K.EQ.18) THEN
    CH = '<'
    RETURN
ENDIF

IF(K.EQ.19) THEN
    CH = '>'
    RETURN
ENDIF

IF(K.EQ.20) THEN
    CH = '?'
    RETURN
ENDIF
END
```

APPENDIX B. A SMALL EXAMPLE

Given the following five classes (pictured in Figure B.1) as candidates for the class containing the true shortest path: We formulate a lower bound for each class.

1. $\alpha_1\alpha_2$
2. $\alpha_2\beta_1$
3. $\alpha_1\beta_2$
4. $\alpha_2\beta_1\alpha_2\beta_2$
5. $\alpha_1\beta_2\alpha_1\beta_1$

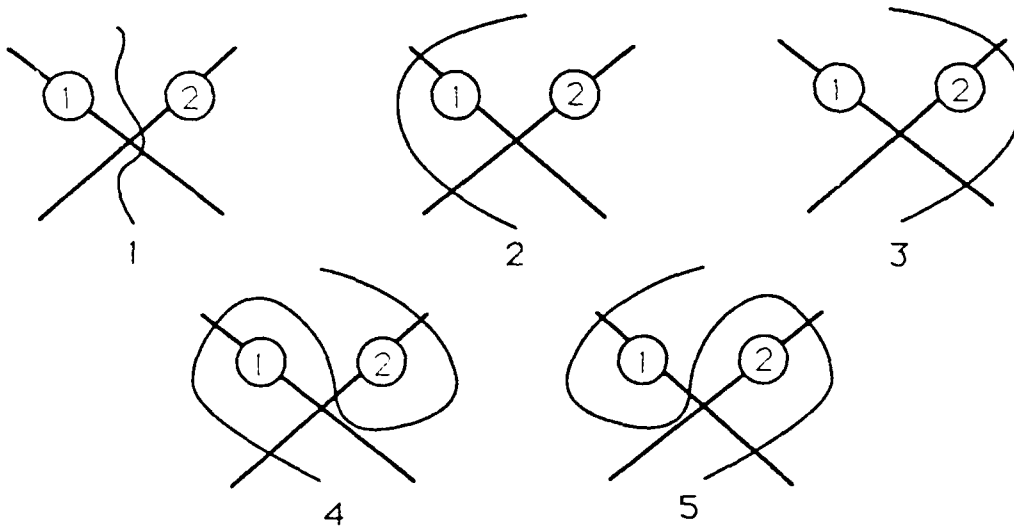


Figure B.1 The Five Classes Associated With Two Obstacles

Consider the class $\alpha_1\beta_2$ pictured below in Figure B.2.

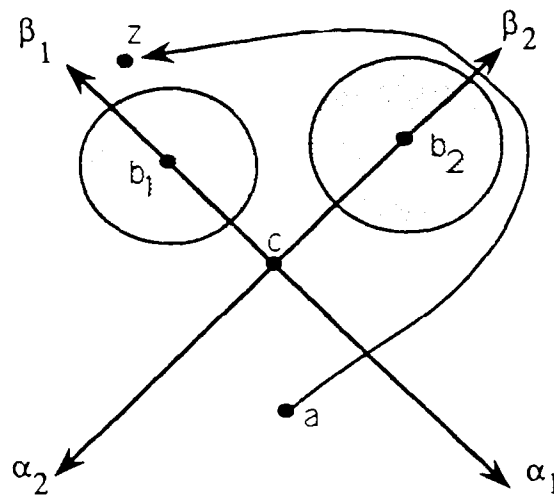


Figure B.2 The Class $\alpha_1\beta_2$

To formulate the lower bound for each class we first shrink the obstacles down to a point (Figure B.3). In order to cross α_2 we develop a cone of directions (Figure B.3) in which the first segment of the bounding polygonal path must lie.

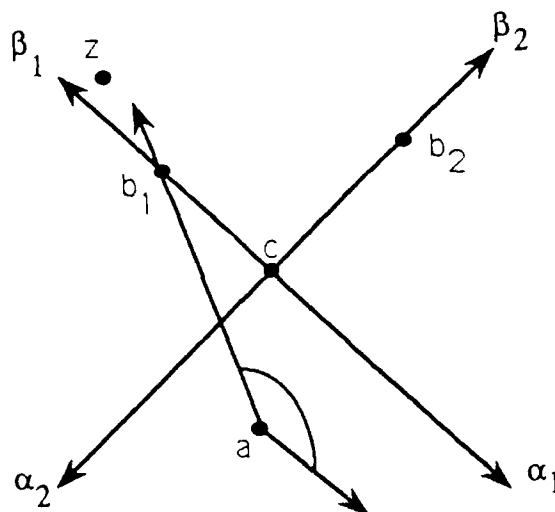


Figure B.3 The Cone of Directions Associated With α_1

This cone is then intersected with the one associated with an β_2 crossing pictured in Figure B.4.

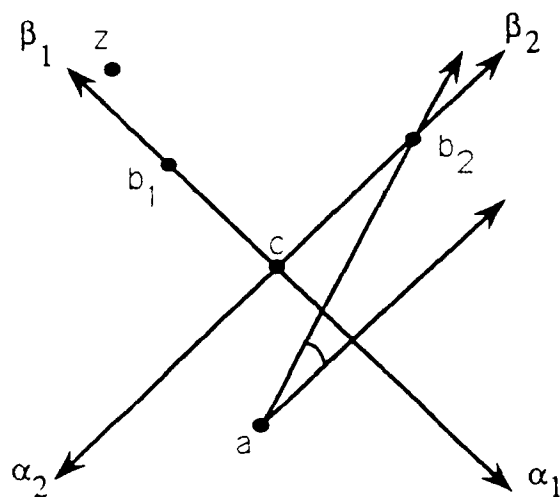


Figure B.4 Cone Associated With β_2

The resulting cone of directions is denoted by the double arc in Figure B.5.

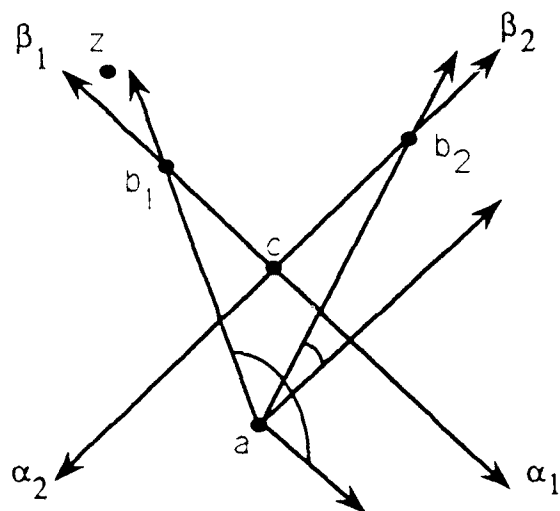


Figure B.5 The Intersected Cone

Since we have reached the end of the class name we check to see if z lies in this cone of directions. If z lies in the cone we go directly to z . Otherwise, we determine which side of the cone z lies so that we can find the peg on which the path bends.

In this example, z does not lie in the final cone of directions so the path bends at the point b_1 . Thus the bounding path is made of two line segments. One from a to b_2 and the other from b_2 to z (Figure B.6).

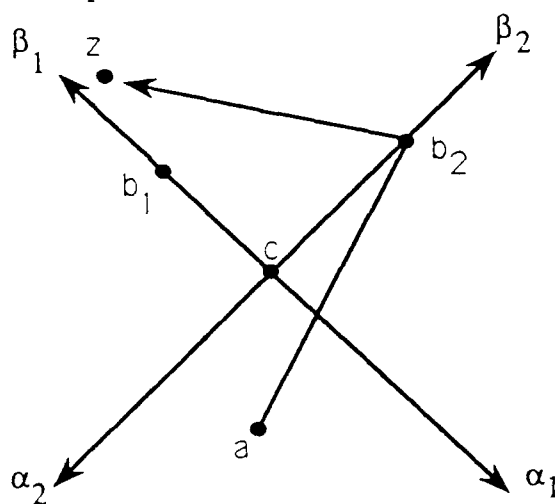


Figure B.6 The Bounding Path for the Class $\alpha_1 \beta_2$

A similar procedure is done on each of the remaining four classes yielding the following lower bounds:

1. $\alpha_1 \alpha_2$: 1.5231
2. $\alpha_2 \beta_1$: 1.5303
3. $\alpha_1 \beta_2$: 2.2885
4. $\alpha_2 \beta_1 \alpha_2 \beta_2$: 3.2885
5. $\alpha_1 \beta_2 \alpha_1 \beta_1$: 2.7008

This list is then put into increasing order:

1. $\alpha_1\alpha_2$: 1.5231
2. $\alpha_2\beta_1$: 1.5303
3. $\alpha_1\beta_2$: 2.2885
4. $\alpha_1\beta_2\alpha_1\beta_1$: 2.7008
5. $\alpha_2\beta_1\alpha_2\beta_2$: 3.2885

Now the first class on this ordered list is chosen. To find the true shortest path we expand the obstacles back to their original shape. If the path bends on a peg, then a tangent line is constructed from a start point to the correct side of the obstacle (Figure B.7).

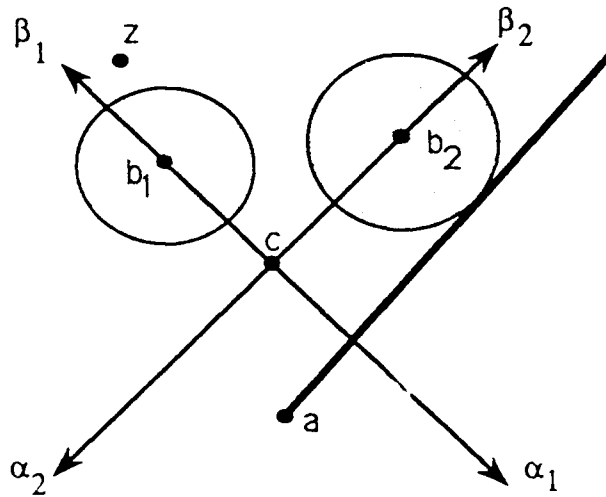


Figure B.7 A Tangent Line From a to B_2

In the class $\alpha_1\beta_2$, we must draw a tangent line from a to b_2 initially. The point of tangency on B_2 can be found by using the formulas developed in Chapter V of this paper. We then travel around the obstacle until a line tangent to B_2 can be drawn to z (Figure B.8).

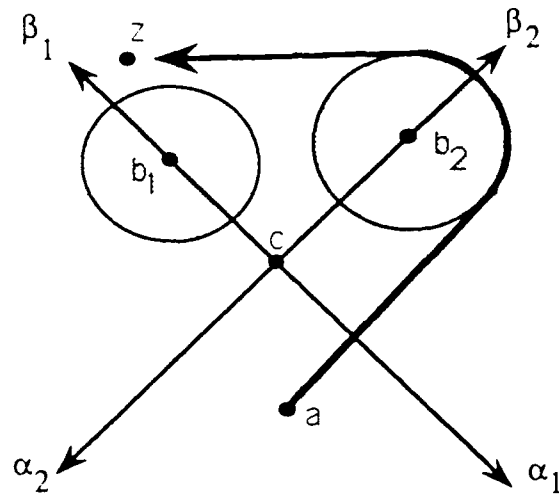


Figure B.8 The Shortest Path in the Class $\alpha_1\beta_2$

Throughout this process the lengths of the line segments and the arc lengths representing the distance traveled around the obstacles is summed. The final sum is the length of the true shortest path for the respective class.

If the shortest path in a class is smaller than the lower bound on the next class to be considered, then the search stops. In this example the search stops after the first class is checked since the length of $\alpha_1\alpha_2$ is 1.8793 and the length of $\alpha_1\beta_1$ is 1.9032.

LIST OF REFERENCES

1. Willard, S., *General Topology*, Addison-Wesley Publishing Company, 1970.
2. Thornton, J. R., *Algebraic Names for Homotopy Classes of Paths in a Plane With Obstacles: A Foundation for the Shortest Path Problem*, NPS Technical Report, Unpublished.
3. Jenkins, K. D., *The Shortest Path Problem in the Plane With Obstacles: A Graph Modeling Approach to Producing Finite Search Lists of Homotopy*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1991.
4. Patterson, E. M., *Topology*, p. 74, Interscience Publishers, Inc., 1963.
5. Thornton, J. R., *Two Problems Concerning Robot Arms: Product-Automation-Based Control Among Obstacle and Recursive Optimization on a Hypercube Compute*, Doctoral Dissertation, Clemson University, Clemson, South Carolina, May 1989

INITIAL DISTRIBUTION LIST

- | | | |
|----|--|---|
| 1. | Defense Technical Information Center
Cameron Station
Alexandria, VA 22304-6145 | 2 |
| 2. | Library, Code 52
Naval Postgraduate School
Monterey, CA 93943-5002 | 2 |
| 3. | Professor John Thornton, Code MA Th
Department of Mathematics
Naval Postgraduate School
Monterey, CA 93943-5000 | 3 |
| 4. | Professor Kim Hefner, Code MA Hk
Department of Mathematics
Naval Postgraduate School
Monterey, CA 93943-5000 | 1 |
| 5. | Chairman, Code MA
Department of Mathematics
Naval Postgraduate School
Monterey, CA 93943-5000 | 1 |
| 6. | CPT Andre M. Cuerington
919 28th Avenue
East Moline, IL 61244 | 3 |
| 7. | CAPT Kevin D. Jenkins
712 Lisburn Road
Camp Hill, PA 17011 | 1 |